Java DB Developer's Guide

Version 10.5

Derby Document build:
August 10, 2009, 10:01:53 AM (EDT)

Contents

Copyright....5

License....6

Relationship between Java DB and Derby....10

About this guide....11
- **Purpose of this guide**....11
- **Audience**....11
- **How this guide is organized**....11

After installing....13
- **The installation directory**....13
  - Batch files and shell scripts....13
- **Derby and JVMs**....13
- **Derby libraries and classpath**....14
- **UNIX-specific issues**....14
  - Configuring file descriptors....14
  - Scripts....14

Upgrades....15
- **Preparing to upgrade**....15
- **Upgrading a database**....15
- **Soft upgrade limitations**....16

JDBC applications and Derby basics....17
- **Application development overview**....17
- **Derby embedded basics**....17
  - Derby JDBC driver....17
  - Derby JDBC database connection URL....18
  - Derby system....19
  - A Derby database....23
  - Connecting to databases....26
  - Working with the database connection URL attributes....29
- **Working with Derby properties**....32
  - Properties overview....32
  - Setting Derby properties....34
  - Properties case study....36

Deploying Derby applications....38
- **Deployment issues**....38
  - Embedded deployment application overview....38
  - Deploying Derby in an embedded environment....39
- **Creating Derby databases for read-only use**....40
  - Creating and preparing the database for read-only use....40
  - Deploying the database on the read-only media....40
  - Transferring read-only databases to archive (jar or zip) files....40
  - Accessing a read-only database in a zip/jar file....41
  - Accessing databases within a jar file using the classpath....42
  - Databases on read-only media and DatabaseMetaData....42
- **Loading classes from a database**....43
  - Class loading overview....43
  - Dynamic changes to jar files or to the database jar classpath....45

Derby server-side programming....46
- **Programming database-side JDBC routines**....46

Database-side JDBC routines and nested connections........46
Database-side JDBC routines using non-nested connections........47
Database-side JDBC routines and SQLExceptions........47
User-defined SQLExceptions........48
Programming trigger actions........48
Trigger action overview........48
Performing referential actions........49
Accessing before and after rows........49
Examples of trigger actions........49
Triggers and exceptions........49
Programming Derby-style table functions........50
Overview of Derby-style table functions........50
Example Derby-style table function........52
Optimizer support for Derby-style table functions........52

Controlling Derby application behavior........57
The JDBC connection and transaction model........57
Connections........57
Transactions........58
Result set and cursor mechanisms........61
Simple non-updatable result sets........61
Updatable result sets........62
Result sets and auto-commit........66
Scrollable result sets........66
Holdable result sets........67
Locking, concurrency, and isolation........68
Isolation levels and concurrency........69
Configuring isolation levels........71
Lock granularity........72
Types and scope of locks in Derby systems........72
Deadlocks........75
Working with multiple connections to a single database........80
Deployment options and threading and connection modes........80
Multi-user database access........81
Multiple connections from a single application........81
Working with multiple threads sharing a single connection........81
Pitfalls of sharing a connection among threads........81
Multi-thread programming tips........82
Example of threads sharing a statement........83
Working with database threads in an embedded environment........83
Working with Derby SQLExceptions in an application........83
Information provided in SQL Exceptions........83

Using Derby as a J2EE resource manager........85
Classes that pertain to resource managers........85
Getting a DataSource........86
Shutting down or creating a database........86

Derby and Security........88
Configuring security for your environment........89
Configuring security in a client/server environment........90
Configuring security in an embedded environment........90
Working with user authentication........90
Enabling user authentication........92
Defining users........92
External directory service........93
Built-in Derby users........97

List of user authentication properties....98
Programming applications for Derby user authentication....99
Users and authorization identifiers....100
Authorization identifiers, user authentication, and user authorization....100
Database owner....101
User names and schemas....101
Exceptions when using authorization identifiers....101
User authorizations....101
Setting the default connection access mode....103
Setting access for individual users....103
Setting the SQL standard authorization mode....104
Encrypting databases on disk....110
Requirements for Derby encryption....111
Working with encryption....111
Signed jar files....116
Notes on the Derby security features....116
User authentication and authorization examples....117
User authentication example in a client/server environment....117
User authentication example in a single-user, embedded environment....123
User authentication examples using SQL authorization....129
Running Derby under a security manager....144
Granting permissions to Derby....144
Examples of Java 2 security policy files for embedded Derby....146

Developing tools and using Derby with an IDE....148
Offering connection choices to the user....148
The DriverPropertyInfo Array....148
Using Derby with IDEs....149
IDEs and multiple JVMs....149

SQL tips....151
Retrieving the database connection URL....151
Supplying a parameter only once....151
Defining an identity column....151
Using third-party tools....151
Tricks of the VALUES clause....152
Multiple rows....152
Mapping column values to return values....152
Creating empty queries....152

Localizing Derby....153
SQL parser support for Unicode....153
Character-based collation in Derby....153
Other components with locale support....155
Messages libraries....156

Derby and standards....157
XML data types and operators....158

Trademarks....159

Apache Software FoundationJava DB Developer's GuideApache Derby

Copyright

Related information

License

License

The Apache License, Version 2.0

```
      that are managed by, or on behalf of, the Licensor for the
      purpose of discussing and improving the Work, but excluding
      communication that is conspicuously marked or otherwise
      designated in writing by the copyright owner as "Not a
      Contribution."

      "Contributor" shall mean Licensor and any individual or Legal
      Entity on behalf of whom a Contribution has been received by
      Licensor and subsequently incorporated within the Work.

   2. Grant of Copyright License. Subject to the terms and conditions
      of this License, each Contributor hereby grants to You a
      perpetual, worldwide, non-exclusive, no-charge, royalty-free,
      irrevocable copyright license to reproduce, prepare Derivative
      Works of, publicly display, publicly perform, sublicense, and
      distribute the Work and such Derivative Works in Source or
      Object form.

   3. Grant of Patent License. Subject to the terms and conditions of
      this License, each Contributor hereby grants to You a perpetual,
      worldwide, non-exclusive, no-charge, royalty-free, irrevocable
      (except as stated in this section) patent license to make, have
      made, use, offer to sell, sell, import, and otherwise transfer
      the Work, where such license applies only to those patent claims
      licensable by such Contributor that are necessarily infringed by
      their Contribution(s) alone or by combination of their
      Contribution(s) with the Work to which such Contribution(s) was
      submitted. If You institute patent litigation against any entity
      (including a cross-claim or counterclaim in a lawsuit) alleging
      that the Work or a Contribution incorporated within the Work
      constitutes direct or contributory patent infringement, then any
      patent licenses granted to You under this License for that Work
      shall terminate as of the date such litigation is filed.

   4. Redistribution. You may reproduce and distribute copies of the
      Work or Derivative Works thereof in any medium, with or without
      modifications, and in Source or Object form, provided that You
      meet the following conditions:

      (a) You must give any other recipients of the Work or
          Derivative Works a copy of this License; and

      (b) You must cause any modified files to carry prominent notices
          stating that You changed the files; and

      (c) You must retain, in the Source form of any Derivative Works
          that You distribute, all copyright, patent, trademark, and
          attribution notices from the Source form of the Work,
          excluding those notices that do not pertain to any part of
          the Derivative Works; and

      (d) If the Work includes a "NOTICE" text file as part of its
          distribution, then any Derivative Works that You distribute
          must include a readable copy of the attribution notices
          contained within such NOTICE file, excluding those notices
          that do not pertain to any part of the Derivative Works, in
          at least one of the following places: within a NOTICE text
          file distributed as part of the Derivative Works; within the
          Source form or documentation, if provided along with the
          Derivative Works; or, within a display generated by the
          Derivative Works, if and wherever such third-party notices
          normally appear. The contents of the NOTICE file are for
          informational purposes only and do not modify the License.
          You may add Your own attribution notices within Derivative
          Works that You distribute, alongside or as an addendum to
          the NOTICE text from the Work, provided that such additional
          attribution notices cannot be construed as modifying the
          License.

      You may add Your own copyright statement to Your modifications
```

```
      and may provide additional or different license terms and
      conditions for use, reproduction, or distribution of Your
      modifications, or for any such Derivative Works as a whole,
      provided Your use, reproduction, and distribution of the Work
      otherwise complies with the conditions stated in this License.

   5. Submission of Contributions. Unless You explicitly state
      otherwise, any Contribution intentionally submitted for
      inclusion in the Work by You to the Licensor shall be under the
      terms and conditions of this License, without any additional
      terms or conditions.  Notwithstanding the above, nothing herein
      shall supersede or modify the terms of any separate license
      agreement you may have executed with Licensor regarding such
      Contributions.

   6. Trademarks. This License does not grant permission to use the
      trade names, trademarks, service marks, or product names of the
      Licensor, except as required for reasonable and customary use
      in describing the origin of the Work and reproducing the content
      of the NOTICE file.

   7. Disclaimer of Warranty. Unless required by applicable law or
      agreed to in writing, Licensor provides the Work (and each
      Contributor provides its Contributions) on an "AS IS" BASIS,
      WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
      implied, including, without limitation, any warranties or
      conditions of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or
      FITNESS FOR A PARTICULAR PURPOSE. You are solely responsible for
      determining the appropriateness of using or redistributing the
      Work and assume any risks associated with Your exercise of
      permissions under this License.

   8. Limitation of Liability. In no event and under no legal theory,
      whether in tort (including negligence), contract, or otherwise,
      unless required by applicable law (such as deliberate and
      grossly negligent acts) or agreed to in writing, shall any
      Contributor be liable to You for damages, including any direct,
      indirect, special, incidental, or consequential damages of any
      character arising as a result of this License or out of the use
      or inability to use the Work (including but not limited to
      damages for loss of goodwill, work stoppage, computer failure or
      malfunction, or any and all other commercial damages or losses),
      even if such Contributor has been advised of the possibility of
      such damages.

   9. Accepting Warranty or Additional Liability. While redistributing
      the Work or Derivative Works thereof, You may choose to offer,
      and charge a fee for, acceptance of support, warranty, indemnity,
      or other liability obligations and/or rights consistent with this
      License. However, in accepting such obligations, You may act only
      on Your own behalf and on Your sole responsibility, not on behalf
      of any other Contributor, and only if You agree to indemnify,
      defend, and hold each Contributor harmless for any liability
      incurred by, or claims asserted against, such Contributor by
      reason of your accepting any such warranty or additional
      liability.

   END OF TERMS AND CONDITIONS

   APPENDIX: How to apply the Apache License to your work.

      To apply the Apache License to your work, attach the following
      boilerplate notice, with the fields enclosed by brackets "[]"
      replaced with your own identifying information. (Don't include
      the brackets!)  The text should be enclosed in the appropriate
      comment syntax for the file format. We also recommend that a
      file or class name and description of purpose be included on the
      same "printed page" as the copyright notice for easier
      identification within third-party archives.
```

Relationship between Java(TM) DB and Derby

Java(TM) DB is a relational database management system that is based on the Java programming language and SQL. Java DB is a commercial release of the Apache Software Foundation's (ASF) open source relational database project. The Apache project is called Derby.

The Java DB product includes Derby without any modification whatsoever to the underlying source code.

Technical support is available for purchase for the Java DB product through Sun.

Because Java DB and Derby have the same functionality, the Java DB documentation refers to the core functionality as Derby.

Java DB Version 10.5 is based on the Derby Version 10.5 release. References to "Derby" in the Java DB documentation refer to the Version 10.5 release of Apache Derby.

About this guide

For general information about the Derby documentation, such as a complete list of books, conventions, and further reading, see *Getting Started with Java DB*.

Purpose of this guide

This guide explains how to use the core Derby technology and is for developers building Derby applications.

It describes basic Derby concepts, such as how you create and access Derby databases through JDBC routines and how you can deploy Derby applications.

Audience

This guide is intended for software developers who already know some SQL and Java.

Derby users who are not familiar with the SQL standard or the Java programming language will benefit from consulting books on those subjects.

How this guide is organized

This document includes the following sections.

- *After installing*

  Explains the installation layout.
- *Upgrades*

  Explains how to upgrade a database created with a previous version of Derby.
- *JDBC applications and Derby basics*

  Basic details for using Derby, including loading the JDBC driver, specifying a database URL, starting Derby, and working with Derby properties.
- *Deploying Derby applications*

  An overview of different deployment scenarios, and tips for getting the details right when deploying applications.
- *Derby server-side programming*

  Describes how to program database-side JDBC routines, triggers, and table functions.
- *Controlling Derby application behavior*

  JDBC, cursors, locking and isolation levels, and multiple connections.
- *Using Derby as a J2EE resource manager*

  Information for programmers developing back-end components in a J2EE system.
- *Derby and Security*

  Describes how to use the security features of Derby.
- *Developing tools and using Derby with an IDE*

  Tips for tool designers.
- *SQL tips*

  Insiders' tricks of the trade for using SQL.
- *Localizing Derby*

An overview of database localization.

- *Derby and standards*

  Describes those parts of Derby that are non-standard or not typical for a database system.

After installing

This section provides reference information about the installation directory, JVMs, classpath, upgrades, and platform-specific issues.

Review the `index.html` file at the top level of the Derby distribution for pointers to reference and tutorial information about Derby. See the Release Notes for information on platform support, changes that may affect your existing applications, defect information, and recent documentation updates. See *Getting Started with Java DB* for basic product descriptions, information on getting started, and directions for setting the path and the classpath.

The installation directory

You may install the Derby software in a directory of your choice.

See the `index.html` file for pointers to information on Derby.

The distribution includes setup scripts that use an environment variable called DERBY_HOME. The variable's value is set to the Derby base directory.

```
C:>echo %DERBY_HOME%
C:\DERBY_HOME
```

If you want to set your own environment, *Getting Started with Java DB* instructs you on setting its value to the directory in which you installed the Derby software.

The distribution for Derby contains all the files you need, including the documentation set, some example applications, and a sample database.

Details about the installation:

- *index.html* in the top-level directory is the top page for the on-line documentation.
- *RELEASE-NOTES.html*, in the top-level Derby base directory, contains important last-minute information. *Read it first*.
- */bin* contains utilities and scripts for running Derby.
- */demo* contains some sample applications, useful scripts, and prebuilt databases.
    - */databases* includes prebuilt sample databases.
    - */programs* includes sample applications.
- */docs* contains the on-line documentation (including this document).
- */javadoc* contains the documented APIs for the public classes and interfaces. Typically, you use the JDBC interface to interact with Derby; however, you can use some of these additional classes in certain situations.
- */lib* contains the Derby libraries.

Batch files and shell scripts

The */bin* directory contains scripts for running some of the Derby tools and utilities. To customize your environment, put the directory first in your path.

These scripts serve as examples to help you get started with these tools and utilities on any platform. However, they may require modification in order to run properly on certain platforms.

Derby and JVMs

Derby is a database engine written completely in Java; it will run in any JVM, version 1.4 or higher.

Derby libraries and classpath

Derby libraries are located in the */lib* subdirectory of the Derby base directory. You must set the classpath on your development machine to include the appropriate libraries.

Getting Started with Java DB explains how to set the classpath in a development environment.

UNIX-specific issues

This section discusses Derby issues specifically related to UNIX platforms.

Configuring file descriptors

Derby databases create one file per table or index. Some operating systems limit the number of files an application can open at one time.

If the default is a low number, such as 64, you might run into unexpected *IOExceptions* (wrapped in `SQLExceptions`). If your operating system lets you configure the number of file descriptors, set this number to a higher value.

Scripts

Your installation contains executable script files that simplify invoking the Derby tools. On UNIX systems, these files might need to have their default protections set to include execute privilege.

A typical way to do this is with the command *chmod +x *.ksh*.

Consult the documentation for your operating system for system-specific details.

Upgrades

To connect to a database created with a previous version of Derby, you must first upgrade that database.

Upgrading involves writing changes to the system tables, so it is not possible for databases on read-only media. The upgrade process:

- marks the database as upgraded to the current release (Version 10.5).
- allows use of new features.

See the release notes for more information on upgrading your databases to this version of Derby.

Preparing to upgrade

Upgrading your database occurs the first time the new Derby software connects to the old database.

Before you connect to the database using the new software:

1. Back up your database to a safe location using Derby online/offline backup procedures.

   For more information on backup, see the *Java DB Server and Administration Guide*.
2. Update your CLASSPATH with the latest jar files.
3. Make sure that there are no older versions of the Derby jar files in your CLASSPATH. You can determine if you have multiple versions of Derby in your CLASSPATH by using the sysinfo tool.

   To use the `sysinfo` tool, execute the following command:

   ```
   java org.apache.derby.tools.sysinfo
   ```

   The `sysinfo` tool uses information found in the Derby jar files to determine the version of any Derby jar in your CLASSPATH. Be sure that you have only one version of the Derby jar files specified in your CLASSPATH.

Upgrading a database

To upgrade a database, you must explicitly request an upgrade the first time you connect to the database with the new version of Derby.

Ensure that you complete the prerequisite steps before you upgrade:

- Back up your database before you upgrade.
- Ensure that only the new Derby jar files are in your CLASSPATH.

When you upgrade the database, you can perform a full upgrade or soft upgrade:

- A full upgrade is a complete upgrade of the Derby database. When you perform a full upgrade, you cannot connect to the database with an older version of Derby and you cannot revert back to the previous version.
- A soft upgrade allows you to run a newer version of Derby against an existing database without having to fully upgrade the database. This means that you can continue to run an older version of Derby against the database. However, if you perform a soft upgrade, certain features will not be available to you until you perform a full upgrade.

1. To upgrade the database, select the type of upgrade that you want to perform:

Type of upgrade	Action
Full upgrade	**Connect to the database using the `upgrade=true` database connection URL attribute. For example:** `jdbc:derby:sample;upgrade=true`
Soft upgrade	**Connect to the database. For example:** `connect 'jdbc:derby:sample'` **In this example, `sample` is a database from a previous version of Derby.**

Soft upgrade limitations

Soft upgrade allows you to run a newer version of Derby against an existing database without having to fully upgrade the database. This means that you can continue to run an older version of Derby against the database.

If you perform a soft upgrade, you will not be able to perform certain functions that are not available in older versions of Derby. Specifically, new features that affect the structure of a database are not available with a soft upgrade. For a list of the new features in a release, see the Release Notes for that release.

To perform a soft upgrade on a database created using an earlier version of Derby, simply connect to the database, as shown in the following example:

```
connect 'jdbc:derby:sample'
```

JDBC applications and Derby basics

This section describes the core Derby functionality. In addition, it details the most basic Derby deployment, Derby embedded in a Java application.

Application development overview

Derby application developers use JDBC, the application programming interface that makes it possible to access relational databases from Java programs.

The JDBC API is part of the Java(TM) 2 Platform, Standard Edition and is not specific to Derby. It consists of the *java.sql* and *javax.sql* packages, which is a set of classes and interfaces that make it possible to access databases (from a number of different vendors, not just Derby) from a Java application.

To develop Derby applications successfully, you will need to learn JDBC. This section does not teach you how to program with the JDBC API.

This section covers the details of application programming that are specific to Derby applications. For example, all JDBC applications typically start their DBMS's JDBC driver and use a connection URL to connect to a database. This chapter gives you the details of how to start Derby's JDBC driver and how to work with Derby's connection URL to accomplish various tasks. It also covers essential Derby concepts such as the Derby system.

You will find reference information about the particulars of Derby's implementation of JDBC in the *Java DB Reference Manual*.

Derby application developers will need to learn SQL. *SQL* is the standard query language used with relational databases and is not tied to a particular programming language. No matter how a particular RDBMS has been implemented, the user can design databases and insert, modify, and retrieve data using the standard SQL statements and well-defined data types. SQL-92 is the version of SQL standardized by ANSI and ISO in 1992; Derby supports entry-level SQL-92 as well as some higher-level features. Entry-level SQL-92 is a subset of full SQL-92 specified by ANSI and ISO that is supported by nearly all major DBMSs today. This chapter does not teach you SQL. You will find reference information about the particulars of Derby's implementation of SQL in the *Java DB Reference Manual*.

Derby implements JDBC that allows Derby to serve as a resource manager in a J2EE compliant system.

Derby embedded basics

This section explains how to use and configure Derby in an embedded environment.

Included in the installation is a sample application program, */demo/programs/simple*, which illustrates how to run Derby embedded in the calling program.

Derby JDBC driver

Derby consists of both the database engine and an embedded JDBC driver. Applications use JDBC to interact with a database. Applications running on JDK 5 or earlier must load the driver in order to work with the database.

In an embedded environment, loading the driver also starts Derby.

The Derby driver class name for the embedded environment is *org.apache.derby.jdbc.EmbeddedDriver*.

In a Java application, you typically load the driver with the static *Class.forName* method or with the *jdbc.drivers* system property. For example:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

```
java -Djdbc.drivers=org.apache.derby.jdbc.EmbeddedDriver applicationClass
```

For detailed information about loading the Derby JDBC driver, see "java.sql.Driver interface" in the *Java DB Reference Manual*.

If your application runs on JDK 6 or higher, you do not need to explicitly load the `EmbeddedDriver`. In that environment, the driver loads automatically.

If your application shuts down Derby or calls the *DriverManager.unload* method, and you then want to reload the driver, call the *Class.forName().newInstance()* method.

Derby JDBC database connection URL

A Java application using the JDBC API establishes a connection to a database by obtaining a *Connection* object.

The standard way to obtain a *`Connection`* object is to call the method *`DriverManager.getConnection`*, which takes a String containing a connection URL (uniform resource locator). A JDBC connection URL provides a way of identifying a database. It also allows you to perform a number of high-level tasks, such as creating a database or shutting down the system.

An application in an embedded environment uses a different connection URL from that used by applications using the Derby Network Server in a client/server environment. See the *Java DB Server and Administration Guide* for more information on the Network Server.

However, all versions of the connection URL (which you can use for tasks besides connecting to a database) have common features:

- you can specify the name of the database you want to connect to
- you can specify a number of attributes and values that allow you to accomplish tasks. For more information about what you can specify with the Derby connection URL, see Database connection examples.

The connection URL syntax is as follows:

```
jdbc:derby:[subsubprotocol:][databaseName][;attribute=value]*
```

Subsubprotocol, which is not typically specified, determines how Derby looks for a database: in a directory, in a class path, or in a jar file. *Subsubprotocol* is one of the following:

- *directory*: The default. Specify this explicitly only to distinguish a database that might be ambiguous with one on the class path.
- *classpath*: Databases are treated as read-only databases, and all *databaseNames* must begin with at least a slash, because you specify them "relative" to the classpath directory. See Accessing databases from the classpath for details.
- *jar*: Databases are treated as read-only databases. *DatabaseNames* might require a leading slash, because you specify them "relative" to the jar file. See Accessing databases from a jar or zip file for details.

  jar requires an additional element immediately before the database name:

```
(pathToArchive)
```

pathToArchive is the path to the jar or zip file that holds the database.

For detailed reference about connection URL attributes and values, see "Setting attributes for the database connection URL" in the *Java DB Reference Manual*.

The following example shows the use of the connection URL:

```
Connection conn = DriverManager.getConnection("jdbc:derby:sample");
```

Derby system

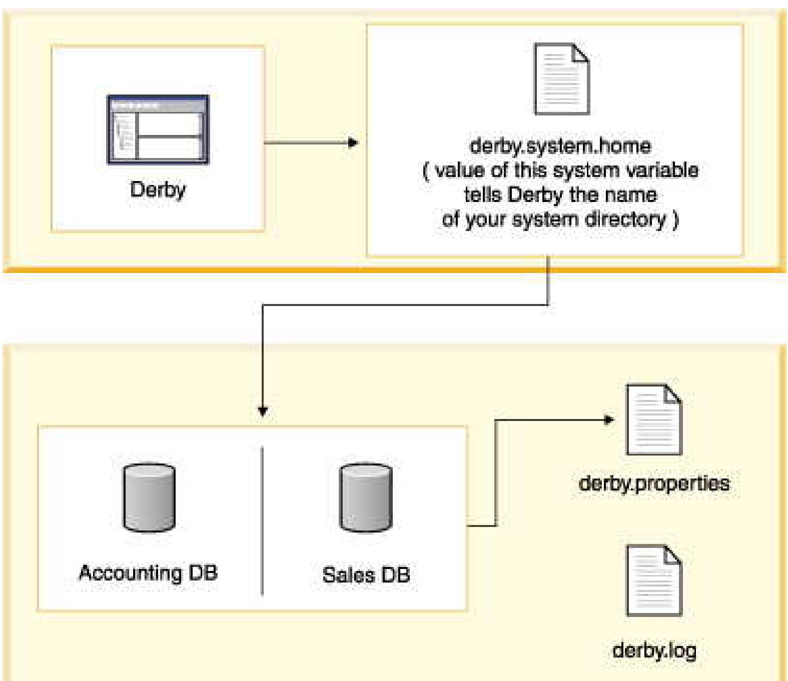
A Derby database exists within a *system*.

A Derby system is a single instance of the Derby database engine and the environment in which it runs. It consists of a system directory, zero or more databases, and a system-wide configuration. The system directory contains any persistent system-wide configuration parameters, or properties, specific to that system in a properties file called derby.properties. This file is not automatically created; you must create it yourself.

The Derby system is not persistent; you must specify the location of the system directory at every startup.

However, the Derby system and the system directory is an essential part of a running database or databases. Understanding the Derby system is essential to successful development and deployment of Derby applications. Derby databases live in a system, which includes system-wide properties, an error log, and one or more databases.

Figure 1. Derby databases live in a system, which includes system-wide properties, an error log, and one or more databases.



The system directory can also contain an error log file called *derby.log* (see The error log).

Each database within that system is contained in a subdirectory, which has the same name as the database (see A Derby database).

In addition, if you connect to a database outside the current system, it automatically becomes part of the current system.

One Derby instance for each Java Virtual Machine

You could potentially have two instances of a Derby system (JVM) running on the same machine at the same time. Each instance must run in a different JVM. Two separate instances of Derby must not access the same database.

For example, in an embedded environment, an application that accesses Derby databases starts up the local JDBC driver, which starts up an instance of Derby. If you start another application, such as ij, and connect to the same database, severe database corruption can result. See Double-booting system behavior.

Booting databases

The default configuration for Derby is to *boot* (or start) a database when an application first makes a connection to it. When Derby boots a database, it checks to see if recovery needs to be run on the database, so in some unusual cases booting can take some time.

You can also configure your system to automatically boot all databases in the system when it starts up; see "*derby.system.bootAll*" in the *Java DB Reference Manual*. Because of the time needed to boot a database, the number of databases in the system directory affects startup performance if you use that configuration.

Once a database has been booted within a Derby system, it remains active until the Derby system has been shut down or until you shut down the database individually.

When Derby boots a database, a message is added to the log file. The message includes the Derby version that the database was booted with, for example:

```
2009-05-08 17:27:11.199 GMT:
 Booting Derby version The Apache Software Foundation - Apache Derby
-  10.5.1.1 - (764942): instance a816c00e-0121-2140-ffd9-fffff0cfee85
on database directory C:\sampledb
```

The number of databases running in a Derby system is limited only by the amount of memory available in the JVM.

Shutting down the system

In an embedded environment, when an application shuts down, it should first shut down Derby.

If the application that started the embedded Derby quits but leaves the Java Virtual Machine (JVM) running, Derby continues to run and is available for database connections.

In an embedded system, the application shuts down the Derby system by issuing the following JDBC call:

```
DriverManager.getConnection("jdbc:derby:;shutdown=true");
```

Shutdown commands always raise *SQLExceptions*.

When a Derby system shuts down, a message goes to the log file:

```
2009-05-08 17:28:47.140 GMT:
Shutting down instance a816c00e-0121-2140-ffd9-fffff0cfee85
```

Typically, an application using an embedded Derby engine shuts down Derby just before shutting itself down. However, an application can shut down Derby and later restart it in

the same JVM session. To restart Derby successfully, the application needs to reload *org.apache.derby.jdbc.EmbeddedDriver* as follows:

```
Class.forName(org.apache.derby.jdbc.EmbeddedDriver).newInstance();
```

Loading the embedded driver starts Derby.

The JDBC specification does not recommend calling `newInstance()`, but adding a `newInstance()` call guarantees that Derby will be booted on any JVM.

It is also possible to shut down a single database instead of the entire Derby system. See Shutting down Derby or an individual database. You can reboot a database in the same Derby session after shutting it down.

Defining the system directory

You define the system directory when Derby starts up by specifying a Java *system property* called *derby.system.home*.

If you do not specify the system directory when starting up Derby, the current directory becomes the system directory.

Derby uses the *derby.system.home* property to determine which directory is its system directory - and thus what databases are in its system, where to create new databases, and what configuration parameters to use. See the *Java DB Reference Manual* for more information on this property.

If you specify a system directory at startup that does not exist, Derby creates this new directory - and thus a new system with no databases-automatically.

The error log

Once you create or connect to a database within a system, Derby begins outputting information and error messages to the error log.

Typically, Derby writes this information to a log called *derby.log* in the system directory, although you can also have Derby send messages to a stream, using the *derby.stream.error.method* property. By default, Derby overwrites *derby.log* when you start the system. You can configure Derby to append to the log with the *derby.infolog.append* property. For information on setting this and other properties, see the *Java DB Reference Manual*.

derby.properties

The text file *derby.properties* contains the definition of properties, or configuration parameters that are valid for the entire system.

The *derby.properties* file is not automatically created. If you want to set Derby properties with this file, you need to create the file yourself. The *derby.properties* file should be in the format created by the *java.util.Properties.save* method. For more information about properties and the *derby.properties* file, see Working with Derby properties and the *Java DB Reference Manual*.

Double-booting system behavior

Derby attempts to prevent two instances of Derby from booting the same database by using a file called *db.lck* inside the database directory.

On all platforms running with a JDK of 1.4 or higher, Derby can successfully prevent a second instance of Derby from booting the database and thus prevents corruption.

On some platforms running with a JDK lower than 1.4, Derby may prevent a second instance of Derby from booting the database (previous to JDK 1.4 the ability to do this was OS dependent).

If this is the case, you will see an *SQLException* like the following:

```
ERROR XJ040: Failed to start database 'sample', see the next exception
 for details.
ERROR XSDB6: Another instance of Derby might have already booted
the databaseC:\databases\sample.
```

The error is also written to the error log.

If you are running a JVM prior to 1.4, Derby issues a warning message on some platforms if an instance of Derby attempts to boot a database that already has a running instance of Derby attached to it. However, it does not prevent the second instance from booting, and thus potentially corrupting, the database. (You can change this behavior with the property `derby.database.forceDatabaseLock`.)

If a warning message has been issued, corruption might already have occurred. Corruption can occur even if one of the two booting systems has "readonly" access to the database.

The warning message looks like this:

```

WARNING: Derby
(instance 80000000-00d2-3265-de92-000a0a0a0200) is
attempting to boot the database /export/home/sky/wombat
even though Derby
(instance 80000000-00d2-3265-8abf-000a0a0a0200) might still be active.
Only one instance of Derby
should boot a database at a time. Severe and non-recoverable corruption
 can
result and might have already occurred.
```

The warning is also written to the error log.

If you see this warning, you should close the connection and exit the JVM, minimizing the risk of a corruption. Close all instances of Derby, then restart one instance of Derby and shut down the database properly so that the *db.lck* file can be removed. The warning message continues to appear until a proper shutdown of the Derby system can delete the *db.lck* file.

When developing applications, you might want to configure Derby to append to the log. Doing so will help you detect when you have inadvertently started more than one instance of Derby in the same system. For example, when the *derby.infolog.append* property is set to true for a system, booting two instances of Derby in the same system produces the following in the log:

```
Sat Aug 14 09:42:51 PDT 2005:
Booting Derby version Apache Derby - 10.0.0.1 - (29612):

instance 80000000-00d2-1c87-7586-000a0a0b1300 on database at
directory C:\tutorial_system\sample
------------------------------------------------------------
Sat Aug 14 09:42:59 PDT 2005:
Booting Derby version Apache Derby - 10.0.0.1 - (29612):
instance 80000000-00d2-1c87-9143-000a0a0b1300 on database at
directory C:\tutorial_system\HelloWorldDB
```

Derby allows you to boot databases that are not in the system directory. While this might seem more convenient, check that you do not boot the same database with two JVMs. If you need to access a single database from more than one JVM, you will need to put a server solution in place. You can allow multiple JVMs that need to access that database to connect to the server. The Derby Network Server is provided as a server solution. See the *Java DB Server and Administration Guide* for more information on the Network Server.

Recommended practices

When developing Derby applications, create a single directory to hold your database or databases.

Give this directory a unique name, to help you remember that:

- All databases exist within a system.
- System-wide properties affect the entire system, and persistent system-wide properties live in the system directory.
- You can boot all the databases in the system, and the boot-up times of all databases affect the performance of the system.
- You can preboot databases only if they are within the system. (Databases do not necessarily have to live inside the system *directory*, but keeping your databases there is the recommended practice.)
- Once you connect to a database, it is part of the current system and thus inherits all system-wide properties.
- Only one instance of Derby can run in a JVM at a single time, and only one instance of Derby should boot a database at one time. Keeping databases in the system directory makes it less likely that you would use more than one instance of Derby.
- The error log is located inside the system directory.

A Derby database

A Derby database contains dictionary objects such as tables, columns, indexes, and jar files. A Derby database can also store its own configuration information.

The database directory

A Derby database is stored in files that live in a directory of the same name as the database. Database directories typically live in *system* directories.

A database directory contains the following, as shown in the following figure.

- *log* directory

  Contains files that make up the database transaction log, used internally for data recovery (not the same thing as the error log).
- *seg0* directory

  Contains one file for each user table, system table, and index (known as conglomerates).
- *service.properties* file

  A text file with internal configuration information.
- *tmp* directory

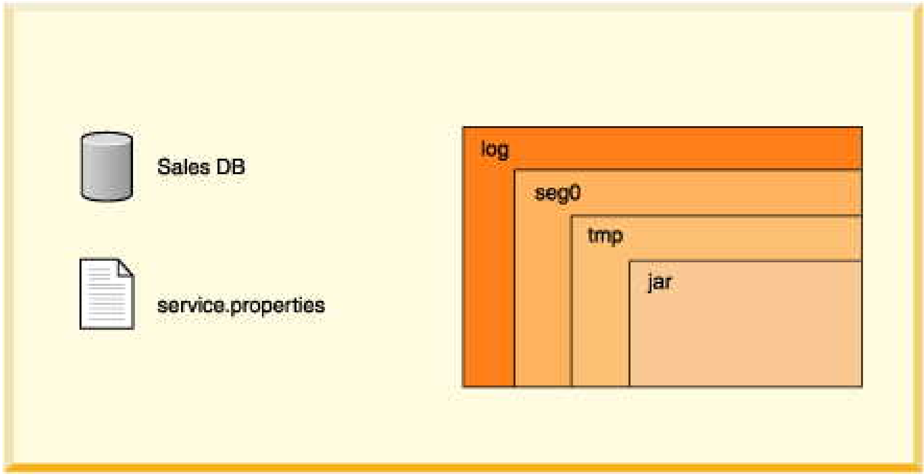
  (might not exist.) A temporary directory used by Derby for large sorts and deferred updates and deletes. Sorts are used by a variety of SQL statements. For databases on read-only media, you might need to set a property to change the location of this directory. See "Creating Derby Databases for Read-Only Use".
- *jar* directory

  (might not exist.) A directory in which jar files are stored when you use database class loading.

Read-only database directories can be archived (and compressed, if desired) into jar or zip files. For more information, see Accessing a read-only database in a zip/jar file.

The following figure shows the files and directories in the Derby database directories that are used by the Derby software.

Figure 2. An example of a Derby database directory and file structure.



Derby imposes relatively few limitations on the number and size of databases and database objects. The following table shows some size limitations of Derby databases and database objects:

Table 1. Size limits for Derby database objects

Type of Object	Limit
tables in each database	*java.lang.Long.MAX_VALUE* Some operating systems impose a limit to the number of files allowed in a single directory.
indexes in each table	32,767 or storage
columns in each table	1,012
number of columns on an index key	16
rows in each table	No limit.
size of table	No limit. Some operating systems impose a limit on the size of a single file.
size of row	No limit. Rows can span pages. Rows cannot span tables so some operating systems impose a limit on the size of a single file, which results in limiting the size of a table and size of a row in that table.

For a complete list of restrictions on Derby databases and database objects, see the *Java DB Reference Manual*.

Creating, dropping, and backing up databases

You create new databases and access existing ones by specifying attributes to the Derby connection URL.

There is no drop database command. To drop a database, delete the database directory with operating system commands. The database must not be booted when you remove a database. You can get a list of booted databases with *getPropertyInfo*.

To back up a database, you can use the online backup utility. For information on this utility, see the *Java DB Server and Administration Guide*.

You can also use *roll-forward recovery* to recover a damaged database. Derby accomplishes roll-forward recovery by using a full backup copy of the database, archived logs, and active logs from the most recent time before a failure. For more information on roll-forward recovery see the *Java DB Server and Administration Guide*.

Single database shutdown

An application can shut down a single database within a Derby system and leave the rest of the system running.

Storage and recovery

A Derby database provides persistent storage and recovery. Derby ensures that all committed transactions are durable, even if the system fails, through the use of a database transaction log.

Whereas inserts, updates, and deletes may be cached before being written to disk, log entries tracking all those changes are never cached but always forced to disk when a transaction commits. If the system or operating system fails unexpectedly, when Derby next starts up it can use the log to perform recovery, recovering the "lost" transactions from the log and rolling back uncommitted transactions. *Recovery* ensures that all committed transactions at the time the system failed are applied to the database, and all transactions that were active are rolled back. Thus the databases are left in a consistent, valid state.

In normal operation, Derby keeps the log small through periodic checkpoints. Checkpointing marks the portions of the log that are no longer useful, writes changed pages to disk, then truncates the log.

Derby checkpoints the log file as it fills. It also checkpoints the log when a shutdown command is issued. Shutting down the JVM in which Derby is running without issuing the proper shutdown command is equivalent to a system failure from Derby's point of view.

Booting a database means that Derby checks to see if recovery needs to be run on a database. Recovery can be costly, so using the proper shutdown command improves connection or startup performance.

Log on separate device

You can put a database's log on a separate device when you create it.

For more information, see the *Java DB Server and Administration Guide*.

Database pages

Derby tables and indexes, known as conglomerates, consist of two or more pages.

A page is a unit of storage whose size is configurable on a system-wide, database-wide, or conglomerate-specific basis. By default, a conglomerate grows one page at a time until eight pages of user data (or nine pages of total disk use, which includes one page of internal information) have been allocated. (You can configure this behavior; see "*derby.storage.initialPages*" in the *Java DB Reference Manual*.) After that, it grows eight pages at a time.

The size of a row or column is not limited by the page size. Rows or columns that are longer than the table's page size are automatically wrapped to overflow pages.

Database-wide properties

You can set many Derby properties as database-level properties. When set in this way, they are stored in the database and "travel" with the database unless overridden by a system property.

For more information, see Scope of properties and Setting database-wide properties.

Derby database limitations

Derby databases have a few limitations.

Indexes

Indexes are not supported for columns defined on CLOB, BLOB, LONG VARCHAR, and XML data types.

If the length of the key columns in an index is larger than half the page size of the index, creating an index on those key columns for the table fails. For existing indexes, an insert of new rows for which the key columns are larger than half of the index page size causes the insert to fail.

Avoid creating indexes on long columns. Create indexes on small columns that provide a quick look-up to larger, unwieldy data in the row. You might not see performance improvements if you index long columns. For information about indexes, see *Tuning Java DB*.

System shutdowns

The system shuts down if the database log cannot allocate more disk space.

A "LogFull" error or some sort of `IOException` occurs in the `derby.log` file when the system runs out of space. If the system has no more disk space to append to the `derby.log` file, you might not see the error messages.

Connecting to databases

You connect to a database using a form of the Derby connection URL as an argument to the *DriverManager.getConnection* call.

You specify a path to the database within this connection URL.

Connecting to databases within the system

The standard way to access databases is in the file system by specifying the path to the database, either absolute or relative to the system directory. In a client/server environment, this path is always on the *server* machine.

By default, you can connect to databases within the current system directory (see Defining the system directory). To connect to databases within the current system, just specify the database name on the connection URL. For example, if your system directory contains a database called *myDB*, you can connect to that database with the following connection URL:

```
jdbc:derby:myDB
```

The full call within a Java program would be:

```
Connection conn =DriverManager.getConnection("jdbc:derby:myDB");
```

Connecting to databases outside the system directory

You can also connect to databases in other directories (including subdirectories of the system directory) by specifying a relative or absolute path name to identify the database. The way you specify an absolute path is defined by the host operating system.

Using the connection URL as described here, you can connect to databases in more than one directory at a time.

Two examples:

```
jdbc:derby:../otherDirectory/myDB
```

```
jdbc:derby:c:/otherDirectory/myDB
```

Note: Once connected, such a database becomes a part of the Derby system, even though it is not in the system directory. This means that it takes on the system-wide properties of the system and no other instance of Derby should access that database. It is recommended that you connect to databases only in the system directory.

Conventions for specifying the database path

When accessing databases from the file system (instead of from classpath or a jar file), any path that is not absolute is interpreted as relative to the system directory.

The path must do one of the following:

- refer to a previously created Derby database
- specify the *create=true* attribute

The path separator in the connection URL is / (forward slash), as in the standard *file://* URL protocol.

You can specify only databases that are local to the machine on which the JVM is running. NFS file systems on UNIX and remote shared files on Windows (*//machine/directory*) are not guaranteed to work. Using *derby.system.home* and forward slashes is recommended practice for platform independent applications.

If two different database name values, relative or absolute, refer to the same actual directory, they are considered equivalent. This means that connections to a database through its absolute path and its relative path are connections to the same database. Within Derby, the name of the database is defined by the canonical path of its directory from *java.io.File.getCanonicalPath*.

Derby automatically creates any intermediate directory that does not already exist when creating a new database. If it cannot create the intermediate directory, the database creation fails.

If the path to the database is ambiguous, i.e., potentially the same as that to a database that is available on the classpath (see "Special Database Access"), use the *directory:* subsubprotocol to specify the one in the file system. For example:

```
jdbc:derby:directory:myDB
```

Special database access

You can also access databases from the classpath or from a jar file (in the classpath or not) as read-only databases.

Accessing databases from the classpath:

In most cases, you access databases from the file system. However, it is also possible to access databases from the classpath. The databases can be archived into a jar or zip file or left as is.

All such databases are read-only.

To access an unarchived database from the classpath, specify the name of the database relative to the directory in the classpath. You can use the classpath subprotocol if such a database is ambiguous within the directory system.

For example, for a database called *sample* in *C:\derby\demo\databases*, you can put the *C:\derby\demo\databases* directory in the classpath and access *sample* like this:

```
jdbc:derby:/sample
```

The forward slash is required before *sample* to indicate that it is relative to *C:\derby\demo\databases* directory.

If only *C:\derby* were in the class path, you could access *sample* (read-only) like this:

```
jdbc:derby:/demo/databases/sample
```

Accessing databases from a jar or zip file:

It is possible to access databases from a jar file. The jar file does not have to be on the classpath.

Note: All such databases are read-only.

For example, suppose you have archived the database *jarDB1* into a file called *jar1.jar*. This archive is in the classpath before you start up Derby. You can access *jarDB1* with the following connection URL

```
jdbc:derby:/jarDB1
```

To access a database in a jar file that is not on the classpath, use the jar subprotocol.

For example, suppose you have archived the database *jarDB2* into a file called *jar2.jar*. This archive is not in the classpath. You can access *jarDB2* by specifying the path to the jar file along with the jar subsubprotocol, like this:

```
jdbc:derby:jar:(c:/derby/lib/jar2.jar)jarDB2
```

For complete instructions and examples of accessing databases in jar files, see Accessing a read-only database in a zip/jar file.

Database connection examples

The examples in this section use the syntax of the connection URL for use in an embedded environment.

This information also applies to the client connection URL in a client/server environment. For reference information about client connection URLs, see "java.sql.Connection interface" in the *Java DB Reference Manual*.

- *jdbc:derby:db1*

  Open a connection to the database *db1*. *db1* is a directory located in the system directory.
- *jdbc:derby:london/sales*

  Open a connection to the database *london/sales*. *london* is a subdirectory of the system directory, and *sales* is a subdirectory of the directory *london*.
- *jdbc:derby:/reference/phrases/french*

  Open a connection to the database */reference/phrases/french*.

  On a UNIX system, this would be the path of the directory. On a Windows system, the path would be *C:\reference\phrases\french* if the current drive were *C.* If a jar file storing databases were in the user's classpath, this could also be a path within the jar file.
- *jdbc:derby:a:/demo/sample*

  Open a connection to the database stored in the directory *\demo\sample* on drive *A* (usually the floppy drive) on a Windows system.
- *jdbc:derby:c:/databases/salesdb jdbc:derby:salesdb*

  These two connection URLs connect to the same database, *salesdb*, on a Windows platform if the system directory of the Derby system is *C:\databases*.
- *jdbc:derby:support/bugsdb;create=true*

Create the database *support/bugsdb* in the system directory, automatically creating the intermediate directory *support* if it does not exist.

- *jdbc:derby:sample;shutdown=true*

  Shut down the *sample* database. (Authentication is not enabled, so no user credentials are required.)
- *jdbc:derby:/myDB*

  Access *myDB* (which is directly in a directory in the classpath) as a read-only database.
- *jdbc:derby:classpath:/myDB*

  Access *myDB* (which is directly in a directory in the classpath) as a read-only database. The reason for using the subsubprotocol is that it might have the same path as a database in the directory structure.
- *jdbc:derby:jar:(C:/dbs.jar)products/boiledfood*

  Access the read-only database *boiledfood* in the *products* directory from the jar file *C:/dbs.jar*.
- *jdbc:derby:directory:myDB*

  Access *myDB*, which is in the system directory. The reason for using the `directory:` subsubprotocol is that it might happen to have the same path as a database in the classpath.

Working with the database connection URL attributes

You specify attributes on the Derby connection URL.

The examples in this section use the syntax of the connection URL for use in an embedded environment. You can also specify these same attributes and values on the client connection URL if you are using Derby as a database server. For more information, see the *Java DB Server and Administration Guide*.

You can also set these attributes by passing a *Properties* object along with a connection URL to `DriverManager.getConnection` when obtaining a connection; see Specifying attributes in a properties object.

All attributes are optional.

For complete information about the attributes, see "Setting attributes for the database connection URL" in the *Java DB Reference Manual*.

For detailed information about the connection URL syntax, see Derby JDBC database connection URL.

Using the databaseName attribute

You can use a databaseName attribute on a database connection URL to specify the name of the database to which you want to connect.

```
jdbc:derby:;databaseName=databaseName
```

You can access read-only databases in jar or zip files by specifying `jar` as the subsubprotocol, like this:

```
jdbc:derby:jar:(pathToArchive)databasePathWithinArchive
```

Or, if the jar or zip file has been included in the classpath, like this:

```
jdbc:derby:/databasePathWithinArchive
```

Shutting down Derby or an individual database

Applications in an embedded environment shut down the Derby system by specifying the *shutdown=true* attribute in the connection URL. To shut down the system, you do not specify a database name, and you must not specify any other attribute.

```
jdbc:derby:;shutdown=true
```

A successful shutdown always results in an *SQLException* to indicate that Derby has shut down and that there is no other exception.

You can also shut down an individual database if you specify the *databaseName*. You can shut down the database of the current connection if you specify the default connection instead of a database name(within an SQL statement).

```
// shutting down a database from your application
DriverManager.getConnection(
    "jdbc:derby:sample;shutdown=true");
```

If user authentication and SQL authorization are both enabled, only the database owner can shut down the database.

```
// shutting down an authenticated database as database owner
DriverManager.getConnection(

 "jdbc:derby:securesample;user=joeowner;password=secret;shutdown=true");
```

Attention: It is good practice to close existing connections before shutting down the system or database. Connections created before the shutdown will not be usable after shutdown is performed. Attempting to access connections after shutdown may cause errors including instances of `NullPointerException` or protocol violations.

Creating and accessing a database

You create a database by supplying a new database name in the connection URL and specifying *create=true*.

Derby creates a new database inside a new subdirectory in the system directory. This system directory has the same name as the new database. If you specify a partial path, it is relative to the system directory. You can also specify an absolute path.

```
jdbc:derby:databaseName;create=true
```

For more details about *create=true*, see *"create=true"* in the *Java DB Reference Manual*.

Providing a user name and password

When user authentication is enabled, an application must provide a user name and password. One way to do this is to use the *user=userName* and *password=userPassword* connection URL attributes.

```
jdbc:derby:sample;user=jill;password=toFetchAPail
```

Creating a database with territory-based collation

By default, Derby uses Unicode codepoint collation. However, you can specify territory-based collation when you create the database.

.

You can use the `collation` and `territory` attributes to specify territory-based collation. This type of collation applies only to user-defined tables. The system tables use the Unicode codepoint collation.

Restriction: The `collation` attribute can be specified only when you create a database. You cannot specify this attribute on an existing database or when you upgrade a database.

To create a database with territory-based collation:

1. Specify the language and country codes for the `territory` attribute, and the TERRITORY_BASED value for the `collation` attribute when you create the database.

   For example:

   ```
   jdbc:derby:MexicanDB;create=true;territory=es_MX;collation=TERRITORY_BASED
   ```

Encrypting a database when you create it

If your environment is configured properly, you can create your database as an encrypted database (one in which the database is encrypted on disk). To do this, you use the *dataEncryption=true* attribute to turn on encryption and the *`bootPassword=key`* attribute or the encryptionKey attribute to specify a key for the encryption.

You can also specify an encryption provider and encryption algorithm other than the defaults with the *`encryptionProvider=providerName`* and *`encryptionAlgorithm=algorithm`* attributes.

```
jdbc:derby:encryptedDB;create=true;dataEncryption=true;
bootPassword=DBpassword
```

Creating an encrypted database with an external key

You can create a database and encrypt the database with an external key.

To create an encrypted database using an external key:

1. Use the *encryptionKey* attribute in the connection URL.

   For example to create the database and encrypt the database `encDB` using an external key, specify this URL:

   ```
   jdbc:derby:encDB;create=true;dataEncryption=true;encryptionAlgorithm=DES/
   CBC/NoPadding;encryptionKey=6162636465666768
   ```

   Attention: If you lose the encryption key you will not be able to boot the database.

Booting an encrypted database

You must specify several attributes in the URL when you boot an encrypted database. You must specify these attributes the first time that you connect to the database within a JVM session, or after you shut the database down within the same JVM session.

To boot an existing encrypted database:

1. The attribute that you specify depends on how the database was originally encrypted:
   - If the database was encrypted using the bootPassword mechanism, specify the *bootPassword* attribute. For example:

     ```
     jdbc:derby:wombat;bootPassword=clo760uds2caPe
     ```
   - If the database was encrypted using an external key, specify the *encryptionKey* attribute. For example:

     ```
     jdbc:derby:flintstone;encryptionAlgorithm=AES/CBC/NoPadding;
     encryptionKey=c566bab9ee8b62a5ddb4d9229224c678
     ```

If the algorithm that was used when the database was created is not the default algorithm, you must also specify the *encryptionAlgorithm* attribute. The default encryption algorithm used by Derby is DES/CBC/NoPadding.

Specifying attributes in a properties object

Instead of specifying attributes on the connection URL, you can specify attributes as properties in a `Properties` object that you pass as a second argument to the `DriverManager.getConnection` method.

For example, to set the user name and password:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver");

Properties p = new Properties();

p.setProperty("user", "sa");
p.setProperty("password", "manager");
p.setProperty("create", "true");

Connection conn = DriverManager.getConnection(
    "jdbc:derby:mynewDB", p);
```

If you are running on JDK 6 or higher, you do not normally need to invoke `Class.forName()`. In that environment, the `EmbeddedDriver` loads automatically. The only exception to this rule is when you need to shut down Derby in the middle of your application and then restart it. To restart Derby, create a new instance of the driver as follows:

```
Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance();
```

Working with Derby properties

This section describes how to use Derby properties. For details on specific properties, see the "Derby properties" section of the *Java DB Reference Manual*.

Properties overview

Derby lets you configure behavior or attributes of a system, a specific database, or a specific *conglomerate* (a table or index) through the use of properties.

Examples of behavior or attributes that you can configure are:

- Whether to authorize users
- Page size of tables and indexes
- Where and whether to create an error log
- Which databases in the system to boot

Scope of properties

You use properties to configure a Derby system, database, or conglomerate.

- *system-wide*

  Most properties can be set on a *system-wide* basis; that is, you set a property for the entire system and all its databases and conglomerates, if this is applicable. Some properties, such as error handling and automatic booting, can be configured only in this way, since they apply to the entire system. (For information about the Derby system, see Derby system.)
- *database-wide*

  Some properties can also be set on a *database-wide* basis. That is, the property is true for the selected database only and not for the other databases in the system unless it is set individually within each of them.

For properties that affect conglomerates, changing the value of such properties affects only conglomerates that are created after the change. Conglomerates created earlier are unaffected.

Note: Database-wide properties are stored in the database and are simpler for deployment, in the sense that they follow the database. Database-wide properties are also recommended for security reasons when you use Derby built-in user authentication (see Derby and Security). System-wide properties can be more practical during the development process.

Persistence of properties

A database-wide property always has persistence. That is, its value is stored in the database. Typically, it is in effect until you explicitly change the property or until you set a system-wide property with precedence over database-wide properties (see Precedence of properties).

To disable or turn off a database-wide property setting, set its value to null. This has the effect of removing the property from the list of database properties and restoring the system property setting, if there is one (and if *derby.database.propertiesOnly* has not been set; see Protection of database-wide properties).

A system-wide property might have persistence, depending on how you set it. If you set it programmatically, it persists only for the duration of the JVM of the application that set it. If you set it in the *derby.properties* file, a property persists until:

- That value is changed and the system is rebooted
- The file is removed from the system and the system is rebooted
- The database is booted outside of that system

Precedence of properties

The search order for properties is:

1. System-wide properties set programmatically (as a command-line option to the JVM when starting the application or within application code)
2. Database-wide properties
3. System-wide properties set in the *derby.properties* file

This means, for example, that system-wide properties set programmatically override database-wide properties and system-wide properties set in the *derby.properties* file, and that database-wide properties override system-wide properties set in the *derby.properties* file.

Protection of database-wide properties:

There is one important exception to the search order for properties described above: When you set the *derby.database.propertiesOnly* property to *true*, database-wide properties cannot be overridden by system-wide properties.

This property ensures that a database's environment cannot be modified by the environment in which it is booted. Any application running in an embedded environment can set this property to `true` for security reasons.

See the "Derby properties" section of the *Java DB Reference Manual* for details on the *derby.database.propertiesOnly* property.

Dynamic versus static properties

Most properties are dynamic; that means you can set them while Derby is running, and their values change without requiring a reboot of Derby. In some cases, this change takes place immediately; in some cases, it takes place at the next connection.

Some properties are static, which means changes to their values will not take effect while Derby is running. You must restart or set them before (or while) starting Derby.

For more information, see Making dynamic or static changes to properties.

Setting Derby properties

This section covers the different ways of setting properties.

Setting system-wide properties

You can set system-wide properties programmatically (as a command-line option to the JVM when starting the application or within application code) or in the text file *derby.properties*.

Changing the system-wide properties programmatically:

You can set properties programmatically -- either in application code before booting the Derby driver or as a command-line option to the Java(TM) Virtual Machine (JVM) when booting the application that starts up Derby. When you set properties programmatically, these properties persist only for the duration of the application. Properties set programmatically are not written to the *derby.properties* file or made persistent in any other way by Derby.

Note: Setting properties programmatically works only for the application that starts up Derby; for example, for an application in an embedded environment or for the application server that starts up a server product. It does not work for client applications connecting to a server that is running.

You can set properties programmatically in the following ways:

- As a parameter to the JVM command line
- Using a Properties object within an application or statement

As a parameter to the JVM command line

You can set system-wide properties as parameters to the JVM command line when you start up the application or framework in which Derby is embedded. To do so, you typically use the *-D* option. For example:

```
java -Dderby.system.home=C:\home\Derby\
    -Dderby.storage.pageSize=8192 JDBCTest
```

Using a Properties object within an application or statement

In embedded mode, your application runs in the same JVM as Derby, so you can also set system properties within an application using a *Properties* object before loading the Derby JDBC driver. The following example sets *derby.system.home* on Windows.

```
Properties p = System.getProperties();
p.setProperty("derby.system.home", "C:\databases\sample");
```

Note: If you pass in a *Properties* object as an argument to the *DriverManager.getConnection* call when connecting to a database, those properties are used as database connection URL attributes, not as properties of the type discussed in this section. For more information, see Connecting to databases and Working with the database connection URL attributes as well as the *Java DB Reference Manual*.

Changing the system-wide properties by using the derby.properties file:

You can set persistent system-wide properties in a text file called *derby.properties*, which must be placed in the directory specified by the *derby.system.home* property. There is one *derby.properties* file per system, not one per database. The file must be created in the system directory. In a client/server environment, that directory is on the server. (For more information about a Derby system and the system directory, see Derby system.)

Derby does *not*:

- Provide this file

- Automatically create this file for you
- Automatically write any properties or values to this file

Instead, you must create, write, and edit this file yourself.

The file should be in the format created by the *java.util.Properties.save* method.

The following is the text of a sample properties file:

```
derby.infolog.append=true
derby.storage.pageSize=8192
derby.storage.pageReservedSpace=60
```

Properties set this way are persistent for the system until changed, until the file is removed from the system, or until the system is booted in some other directory (in which case Derby would be looking for *derby.properties* in that new directory). If a database is removed from a system, system-wide properties do not "travel" with the database unless explicitly set again.

Verifying system properties:

You can find out the value of a system property if you set it programmatically. You cannot find out the value of a system property if you set it in the *derby.properties* file.

For example, if you set the value of the `derby.storage.pageSize` system-wide property in your program or on the command line, the following code will retrieve its value from the System Properties object:

```
Properties sprops = System.getProperties();
System.out.println("derby.storage.pageSize value: "
+ sprops.getProperty("derby.storage.pageSize"));
```

You can also use Java Management Extensions (JMX) technology to obtain system information, including some settings that correspond to system properties. For details, visit the wiki page http://wiki.apache.org/db-derby/DerbyJMX and refer to the API documentation for the packages `org.apache.derby.mbeans` and `org.apache.derby.mbeans.drda`. For information on JMX technology, see http://java.sun.com/javase/technologies/core/mntr-mgmt/javamanagement/.

Setting database-wide properties

Database-wide properties, which affect a single database, are stored within the database itself. This allows different databases within a single Derby system to have different properties and ensures that the properties are correctly retained when a database is moved away from its original system or copied.

You should use database-wide properties wherever possible for ease of deployment and for security.

You set and verify database-wide properties using system procedures within SQL statements.

To set a property, you connect to the database, create a statement, and then use the `SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY` procedure, passing the name of the property and the value.

To check the current value of a property, you connect to the database, create a statement, and then use the `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function, passing in the name of the property.

If you specify an invalid value, Derby uses the default value for the property. (If you call the `SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY` function, however, it displays the invalid value.)

See the *Java DB Reference Manual* for more information on how to use these system functions and procedures.

Setting properties in a client/server environment

In a client/server environment, you must set the system properties for the *server*'s system. That means that when you are using the *derby.properties* file, the file exists in the *server*'s *derby.system.home* directory. Client applications can set database-wide properties because they are set via SQL statements.

Table 2. Summary of ways to set properties

Type of property	How you set it
System-wide	• In *derby.properties* • As a command-line option when starting the JVM that holds the server or, if the server is started from within a program, programmatically by the program that hosts the server
Database-wide	Using system procedures and functions in an SQL statement

Making dynamic or static changes to properties

Note: Properties set in the *derby.properties* file and on the command line of the application that boots Derby are *always* static, because Derby reads this file and those parameters only at startup.

Only properties set in the following ways have the potential to be dynamic:

- As database-wide properties
- As system-wide properties via a *Properties* object in the application in which the Derby engine is embedded

See the "Derby properties" section of the *Java DB Reference Manual* for information about specific properties.

Properties case study

Derby allows you a lot of freedom in configuring your system. This freedom can be confusing if you do not understand how properties work. You also have the option of not setting any properties and instead using the Derby defaults, which are tuned for a single-user embedded system.

Imagine the following scenario of an embedded environment:

Your system has a *derby.properties* file, a text file in the system directory, which you have created and named *system_directory*. Your databases have also been created in this directory. The properties file sets the following property:

```
derby.storage.pageSize=8192
```

You start up your application, being sure to set the *derby.system.home* property appropriately:

```
java -Dderby.system.home=c:\system_directory MyApp
```

The command lines in this example assume that you are using a Windows system.

You then create a new table:

```
CREATE TABLE table1 (a INT, b VARCHAR(10))
```

Derby takes the page size of 8192 from the system-wide properties set in the *derby.properties* file, since the property has not been set any other way.

You shut down and then restart your application, setting the value of *derby.storage.pageSize* to 4096 programmatically, as a parameter to the JVM command line:

```
java -Dderby.system.home=c:\system_directory
    -Dderby.storage.pageSize=4096 MyApp

CREATE TABLE anothertable (a INT, b VARCHAR(10))
```

The page size for the *anothertable* table will be 4096 bytes.

You establish a connection to the database and set the value of the page size for all new tables to 32768 as a database-wide property:

```
CallableStatement cs =
  conn.prepareCall("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(?, ?)");
cs.setString(1, "derby.storage.pageSize");
cs.setString(2, "32768");
cs.execute();
cs.close();
```

You then create a new table that automatically inherits the page size set by the property:

```
CREATE TABLE table2 (a INT, b VARCHAR(10))
```

The page size for the *table2* table is 32768 bytes.

You shut down the application, then restart, this time forgetting to set the system-wide property programmatically (as a command-line option to the JVM):

```
java -Dderby.system.home=c:\system_directory MyApp
```

You then create another table:

```
CREATE TABLE table4 (a INT, b VARCHAR(10))
```

Derby uses the persistent database-wide property of 32768 for this table, since the database-wide property set in the previous session is persistent and overrides the system-wide property set in the *derby.properties* file.

What you have is a situation in which three different tables each get a different page size, even though the *derby.properties* file remained constant.

If you remove the *derby.properties* file from the system or remove the database from its current location (forgetting to move the file with it), you could get yet another value for a new table.

To avoid this situation, be consistent in the way you set properties.

Deploying Derby applications

Typically, once you have developed a Derby application and database, you package up the application, the Derby libraries, and the database in some means for distribution to your users. This process is called *deployment*.

This section discusses issues for deploying Derby applications and databases.

Deployment issues

This section discusses deployment options and details.

Embedded deployment application overview

In an embedded environment, Derby runs in the same JVM as the application.

The application can be a single-user application or a multi-user application server. In the latter case, Derby runs embedded in the user-provided server framework, and any client applications use user-provided connectivity or allow the application server to handle all database interaction.

Figure 3. Derby embedded in a single-user Java application

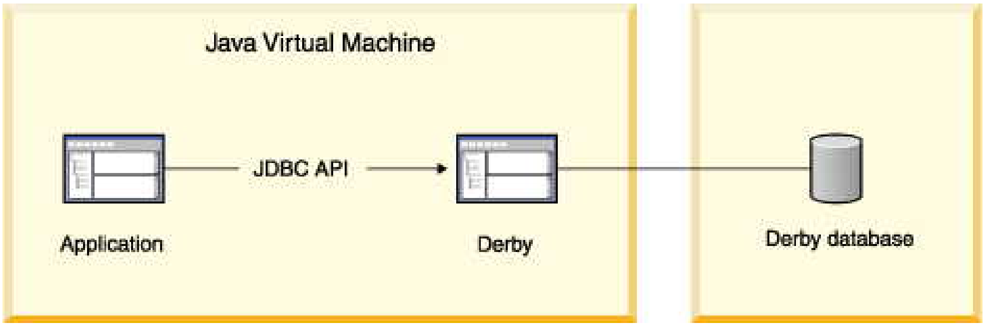
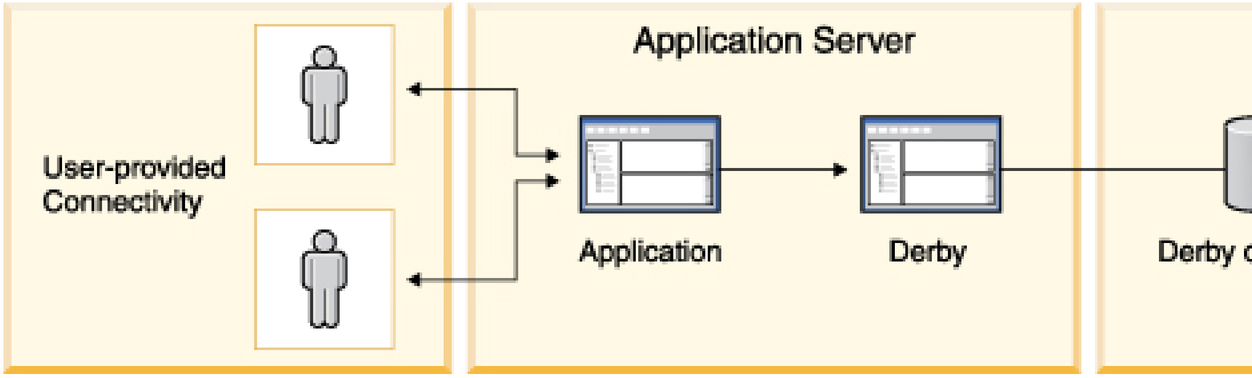


Figure 4. Derby embedded in a multi-user Java application server



When a Derby database is embedded in a Java application, the database is dedicated to that single application. If you deploy more than one copy of the application, *each application has its own copy of the database and Derby software*. A Derby server framework can work in multi-threaded, multi-connection mode and can even connect to more than one database at a time. A server framework, such as the Derby Network

Server, can be used to manage multiple connections and handle network capabilities. Some server framework solutions, such as WebSphere Application Server, provide additional features such as web services and connection pooling. However, only one server framework at a time can operate against a Derby database.

The Derby application accesses an embedded Derby database through the JDBC API. To connect, an application makes a call to the local Derby JDBC driver. Accessing the JDBC driver automatically starts the embedded Derby software. The calling application is responsible for shutting down the embedded Derby database software.

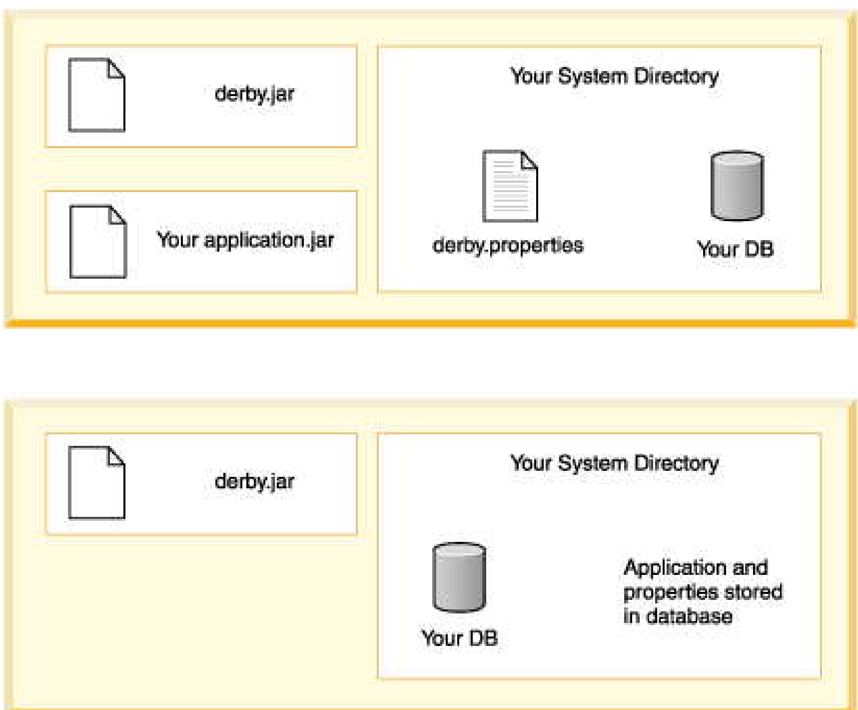
Deploying Derby in an embedded environment

You can embed Derby in any Java application (single- or multi-user) by deploying the following packages.

- The Derby library (*derby.jar*).
- The libraries for the application. You have the option of storing these libraries in the database.
- The database or databases used by the application, in the context of their system directory.

In the following figure, the top graphic shows the deployment of an application, where the application, the Derby software for embedded use, the derby.properties file and the database are four objects. The bottom graphic shows a simplified deployment by reducing the number of objects to two by storing the application and the properties file in the database.

Figure 5. Two approaches to deploying a Derby application in an embedded environment.



Embedded systems and properties

Database-wide properties are stored in the database and are simpler for deployment, while system-wide parameters might be easier for development.

- If you are setting any system-wide properties, see if they can be set as database-wide properties instead.
- Are any properties being set in the *derby.properties* file? Some properties can only be set on a system-wide basis. If so, deploy the entire system directory along with the properties file. Deploy only those databases that you wish to include. Setting properties programmatically can simplify this step- you will not have to worry about deploying the system directory/properties file.

Extra steps are required for deploying an application and an embedded database on read-only media.

Creating Derby databases for read-only use

You can create Derby databases for use on read-only media such as CD-ROMs.

Derby databases in zip or jar files are also read-only databases. Typically, read-only databases are deployed with an application in an embedded environment.

Creating and preparing the database for read-only use

To create databases for use on read-only media, perform these steps.

1. Create and populate the database on read-write media.
2. Commit all transactions and shut down Derby in the prescribed manner. If you do not shut down Derby in the prescribed manner, Derby will need to perform recovery the next time the system boots. Derby cannot perform recovery on read-only media.
3. Delete the *tmp* directory if one was created within your database directory. If you include this directory, Derby will attempt to delete it and will return errors when attempting to boot a database on read-only media.
4. For the read-only database, set the property *derby.storage.tempDirectory* to a writable location.

   Derby needs to write to temporary files for large sorts required by such SQL statements as ORDER BY, UNION, DISTINCT, and GROUP BY. For more information about this property, see the *Java DB Reference Manual*.

   ```
   derby.storage.tempDirectory=c:/temp/mytemp
   ```
5. Configure the database to send error messages to a writable file or to an output stream.

   For information on the *derby.stream.error.file* property, see the *Java DB Reference Manual*.

   ```
   derby.stream.error.file=c:/temp/mylog.LOG
   ```

Be sure to set these properties so that they are deployed with the database.

Deploying the database on the read-only media

To deploy the database on read-only media, perform the following steps.

1. Move the database directory to the read-only media, including the necessary subdirectory directories (*log* and *seg0*) and the file `service.properties`.
2. Use the database as usual, except that you will not be able to insert or update any data in the database or create or drop dictionary objects.

Transferring read-only databases to archive (jar or zip) files

Once a database has been created in Derby, it can be stored in a jar or zip file and continue to be accessed by Derby in read-only mode.

This allows a read-only database to be distributed as a single file instead of as multiple files within a directory and to be compressed. In fact, a jar or zip file can contain any number of Derby databases and can also contain other information not related to Derby, such as application data or code.

You cannot store the *derby.properties* file in a jar or zip file.

To create a jar or zip file containing one or more Derby databases:

1. Create a database for use on read-only media.
2. From the directory that contains the database folder, archive the database directory and its contents. For example, for the database *sales* that lives in the system directory *C:\london*, issue the command from *london*. Do not issue the command from inside the database directory itself.

For example, archive the database folder and its contents using the JAR program from the JDK. You can use any zip or jar tool to generate the archive.

This command archives the database directory *sales* and its contents into a compressed jar file called *dbs.jar*.

```
cd C:\london
jar cMf C:\dbs.jar sales
```

You can add multiple databases with jar. For example, this command puts the *sales* databases and the *boiledfood* database (in the subdirectory products) into the archive.

```
cd C:\london
jar cMf C:\dbs.jar sales products\boiledfood
```

The relative paths of the database in the jar need not match their original relative paths. You can do this by allowing your archive tool to change the path, or by moving the original databases before archiving them.

The archive can be compressed or uncompressed, or individual databases can be uncompressed or compressed if your archive tool allows it. Compressed databases take up a smaller amount of space on disk, depending on the data loaded, but are slower to access.

Once the database is archived into the jar or zip file, it has no relationship to the original database. The original database can continue to be modified if desired.

Accessing a read-only database in a zip/jar file

To access a database in a zip/jar, you specify the jar in the subsubprotocol.

```
jdbc:derby:jar:(pathToArchive)databasePathWithinArchive
```

The *pathToArchive* is the absolute path to the archive file. The *databasePathWithinArchive* is the relative path to the database within the archive. For example:

```
jdbc:derby:jar:(C:/dbs.jar)products/boiledfood
jdbc:derby:jar:(C:/dbs.jar)sales
```

If you have trouble finding a database within an archive, check the contents of the archive using your archive tool. The *databasePathWithinArchive* must match the one in the archive. You might find that the path in the archive has a leading slash, and thus the URL would be:

```
jdbc:derby:jar:(C:/dbs.jar)/products/boiledfood
```

Databases in a jar or zip file are always opened read-only and there is currently no support to allow updates of any type.

Accessing databases within a jar file using the classpath

Once an archive containing one or more Derby databases has been created it can be placed in the classpath. This allows access to a database from within an application without the application's knowing the path of the archive.

When jar or zip files are part of the classpath, you do not have to specify the jar subsubprotocol to connect to them.

To access a database in a zip or jar file in the classpath:

1. Set the classpath to include the jar or zip file before starting up Derby:

   ```
   CLASSPATH="C:\dbs.jar;%CLASSPATH%"
   ```
2. Connect to a database within the jar or zip file with one of the following connection URLs:

   ```
   Standard syntax:
   jdbc:derby:/databasePathWithinArchive
   ```

   ```
   Syntax with subsubprotocol:
   jdbc:derby:classpath:/databasePathWithinArchive
   ```

For example:

```
jdbc:derby:/products/boiledfood
jdbc:derby:classpath:/products/boiledfood
```

Connecting to databases with ambiguous paths to databases in the file system

Use the basic connection URL syntax only if the database path specified does not also point to a Derby database in the file system.

If this is the case, the connection attempt might fail or connect to the wrong database. Use the form of the syntax with the subsubprotocol to distinguish between the databases.

For example:

```
jdbc:derby:classpath:/products/boiledfood
```

Connecting to databases when the path is ambiguous because of databases in the classpath

To connect to a database in the file system when the connection URL that you would use would be ambiguous with a database in the classpath, use the following form of the connection URL.

```
jdbc:derby:directory:databasePathInFileSystem
```

For example,

```
jdbc:derby:directory:/products/boiledfood
```

Apart from the connection URL, databases in archives in the classpath behave just like databases in archives accessed through the file system. However, databases in archives are read-only.

Databases on read-only media and DatabaseMetaData

Databases on read-only media return true for *DatabaseMetaData.isReadOnly*.

Loading classes from a database

You can store application logic in a database and then load classes from the database.

Application logic, which can be used by SQL functions and procedures, includes Java class files and other resources. Storing application code simplifies application deployment, since it reduces the potential for problems with a user's classpath.

In an embedded environment, when application logic is stored in the database, Derby can access classes loaded by the Derby class loader from stored jar files.

Class loading overview

You store application classes and resources by storing one or more jar files in the database. Then your application can access classes loaded by Derby from the jar file and does not need to be coded in a particular way. The only difference is the way in which you invoke the application.

Here are the basic steps.

Create jar files for your application

Include any Java classes in a jar file that are intended for Derby class loading, except the following classes:

- The standard Java packages (*java.**, `javax.*`)

  Derby does not prevent you from storing such a jar file in the database, but these classes *are never loaded* from the jar file.
- The classes that are supplied with your Java environment (for example, *sun.**)

A running Derby system can load classes from any number of jar files from any number of schemas and databases.

Create jar files intended for Derby database class loading the same way you create a jar file for inclusion in a user's classpath. For example, consider an application targeted at travel agencies:

```
jar cf travelagent.jar travelagent/*.class.
```

Various IDEs have tools to generate a list of contents for a jar file based on your application. If your application requires classes from other jar files, you have a choice:

- *Extract the required third-party classes from their jar file and include only those classes in your jar file.*

  Use this option when you need only a small subset of the classes in the third-party jar file.
- *Store the third-party jar file in the database.*

  Use this option when you need most or all of the classes in the third-party jar file, since your application and third-party logic can be upgraded separately.
- *Deploy the third-party jar file in the user's class path.*

  Use this option when the classes are already installed on a user's machine (for example, Objectspace's JGL classes).

Add the jar file or files to the database

Use a set of procedures to install, replace, and remove jar files in a database. When you install a jar file in a database, you give it a Derby jar name, which is an `SQL92Identifier`.

Note: Once a jar file has been installed, you cannot modify any of the individual classes or resources within the jar file. Instead, you must replace the entire jar file.

Jar file examples:

See the *Java DB Tools and Utilities Guide* for reference information about the utility and complete syntax.

Installing jar files:

```
-- SQL  statement
CALL sqlj.install_jar(
    'tours.jar', 'APP.Sample1', 0)

-- SQL  statement
-- using a quoted identifier for the
-- Derby jar name
CALL sqlj.install_jar(
    'tours.jar', 'APP."Sample2"', 0)
```

Removing jar files:

```
-- SQL  statement
CALL sqlj.remove_jar(
    'APP.Sample1', 0)
```

Replacing jar files:

```
-- SQL  statement
CALL sqlj.replace_jar(
    'c:\myjarfiles\newtours.jar', 'APP.Sample1')
```

Enable database class loading with a property

Once you have added one or more jar files to a database, you must set the database jar "classpath" by including the jar file or files in the *derby.database.classpath* property to enable Derby to load classes from the jar files.

This property, which behaves like a class path, specifies the jar files to be searched for classes and resources and the order in which they are searched. If Derby does not find a needed class stored in the database, it can retrieve the class from the user's classpath. (Derby first looks in the user's classpath before looking in the database.)

- Separate jar files with a colon (:).
- Use two-part names for the jar files (schema name and jar name). Set the property as a database-level property for the database. The *first time* you set the property, you must reboot to load the classes.

Example:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.classpath',
    'APP.ToursLogic:APP.ACCOUNTINGLOGIC')
```

See "*derby.database.classpath*" in the *Java DB Reference Manual* for more information about the property.

Note: Derby's class loader looks first in the user's classpath for any needed classes, and then in the database. To ensure class loading with the database class loader, remove classes from the classpath.

Code your applications

In your applications, you load the classes either by indirectly referencing them in the code or by directly using *java.lang.Class.forName*.

You load resources the way you normally would, using the standard *java.lang.Class.getResourceAsStream*, a mechanism that allows an application to access resources defined in the classpath without knowing where or how they are stored.

You do not need to make any changes to the way code interacts with Derby and its JDBC driver. An application can safely attempt to boot Derby, even though it is already running, without any errors. Applications connect to Derby in the usual manner.

Note: The method *getResource* is not supported.

Dynamic changes to jar files or to the database jar classpath

When you store jar files in a single database and make those jar files available to that database, it is possible to make changes to jar files or to change the database jar "classpath" dynamically (without having to reboot).

That is, when you install or replace a jar file within an SQL statement or change the database jar "classpath" (the *derby.database.classpath* property`),` Derby is able to load the new classes right away without your having to reboot.

Requirements for dynamic changes

Certain conditions must be met for Derby to be able to load the new classes right away without you having to reboot.

- You *originally configured* database-level class loading for the database correctly. Turning on the database-level class loading property requires setting the `derby.database.classpath` property with valid two-part names, then rebooting.
- If changes to the *derby.database.classpath* property are needed to reflect new jar files, you change the property to a valid value.

If these requirements are not met, you will have to reboot to see the changes.

Notes on dynamic changes

When you are changing the *derby.database.classpath* property, all classes loaded from database jar files are reloaded, even for a jar file that has not changed.

Remember that the user's classpath is searched first.

Any existing prepared statements will use the previously loaded classes unless they require class loading, in which case they will fail with a *ClassNotFound* error.

Cached objects do not match objects created with newly loaded classes. For example, an in-memory *Customer* object will not match a new `Customer` object if the `Customer` class has been reloaded, and it will raise a `ClassCastException.`

Derby server-side programming

This section discusses special programming for Derby.

In particular, this section discusses how to program database-side JDBC routines, triggers, and table functions.

Programming database-side JDBC routines

Methods invoked within an application are called application-side methods. Methods invoked within Derby are called database-side routines.

An application-side method can be exactly the same as a database-side routine. The only difference is where you invoke them. You write the method only once. Where you invoke the method--within the application or within an SQL statement--determines whether it is an "application-side" or a "database-side" method.

Database-side JDBC routines and nested connections

Most database-side JDBC routines need to share the same transaction space as the statements that called them.

The reasons for this are:

- to avoid blocking and deadlocks
- to ensure that any updates done from within the routine are atomic with the outer transaction

In order to use the same transaction, the routine must use the same connection as the parent SQL statement in which the routine was executed. Connections re-used in this way are called *nested connections*.

Use the connection URL *jdbc:default:connection* to re-use the current *Connection*.

The database donnection URL *jdbc:default:connection* allows a Java method to get the *Connection* of the SQL statement that called it. This is the standard (SQL standard, Part 13, SQL Routines and Java) mechanism to obtain the nested connection object. The method would get a *Connection* as follows:

```
Connection conn = DriverManager.getConnection(
    "jdbc:default:connection");
```

URL attributes are not supported as part of this connection URL. Any URL attributes specified in a Properties object, user name, or password that are passed to a *java.sql.DriverManager.getConnection()* call will be ignored.

Loading a JDBC driver in a database-side routine is not required.

Requirements for database-side JDBC routines using nested connections

In order to preserve transactional atomicity, database-side JDBC routines that use nested connections have the following limitations.

- Can issue a commit or rollback only within a procedure (not a function).
- Cannot change the auto-commit connection attribute.
- Cannot modify the data in a table used by the parent statement that called the routine, using INSERT, UPDATE, or DELETE. For example, if a SELECT statement using the *T* table calls the `changeTables` procedure, `changeTables` cannot modify data in the *T* table.

- Cannot drop a table used by the statement that called the routine.
- Cannot be in a class whose static initializer executes DDL statements.

In addition, the Connection object that represents the nested connection always has its auto-commit mode set to false.

Database-side JDBC routines using non-nested connections

A database-side JDBC routine can create a new connection instead of using a nested connection. Statements executed in the routine will be part of a different transaction, and so can issue commits and rollbacks.

Such a routine can connect to a database different from the one to which the parent SQL statement that called it is connected. The routine does not use the same transaction or *Connection*. It establishes a new *Connection* and transaction.

Note: If database-side JDBC routines do not use nested connections, this means that they are operating outside of the normal DBMS transaction control, so it is not good practice to use them indiscriminately.

Invoking a procedure using the CALL command

If a procedure uses only IN parameters, Derby can execute the procedure by using the SQL CALL command. A stored procedure with IN, OUT, or INOUT parameters can be invoked from a client application by using a *CallableStatement*.

You can invoke the procedure in an SQL statement such as the following:

```
CALL MYPROC()
```

Note: You can roll back a CALL statement only if no commits or rollbacks occur within the specified procedure.

You can also use the CALL command to execute a routine that returns a value, but you will not be able to access the value.

Database-side JDBC routines and SQLExceptions

It is possible to code database-side routines, like application-side methods, to catch *SQLExceptions*. SQLExceptions that are caught within a routine are hidden from the calling application code.

When such *SQLExceptions* are of transaction severity (such as deadlocks), this "hiding" of the exception causes unexpected problems.

This is because errors of transaction severity roll back work already done by a transaction (not just the piece executed by the called method) and silently begin a new transaction. When the method execution is complete, Derby detects that the outer statement was invalidated by a deadlock and rolls back any work done *in the new transaction* as well. This is the expected behavior, because all the statements in between explicit commits should be treated atomically; the new transaction implicitly begun by Derby's rollback was not intended by the application designer.

However, this is not the same behavior that would happen if the method were invoked in the application. In that situation, Derby would roll back the work done by the transaction and silently begin a new transaction. Work in the new transaction would not be rolled back when the method returned. However, coding the application in that way means that the transaction did not end where you expected it to and is probably a programming mistake. Coding in this manner is not recommended.

A method that catches a deadlock exception and then continues is probably making a mistake. Errors of transaction severity should be caught not by nested code, but only by

the outermost application code. That is the only way to ensure that transactions begin and end where you expect them to.

Not all database vendors handle nested deadlocks the same way. For this and other reasons, it is not possible to write portable SQL-invoking methods. However, it is possible to write SQL-invoking methods that behave identically *regardless of whether you invoke them in the application or as a routine in the database*.

In order to ensure identical application- and database-side handling of nested errors, code try-catch blocks to check for the severity of exceptions as follows:

```
try {
    preparedStatement.execute();
} catch (SQLException se ) {
        String SQLState = se.getSQLState();
    if ( SQLState.equals( "23505" ) )
        { correctDuplicateKey(); }
    else if ( SQLState.equals( "22003" ) ) {
        correctArithmeticOverflow(); }
    else { throw se; }
}
```

Of course, users also have the choice of not wrapping SQL statements in try-catch blocks within methods. In that case, *SQLExceptions* are caught higher up in their applications, which is the desired behavior.

User-defined SQLExceptions

When the execution of a database-side method raises an error, Derby wraps that exception in an *SQLException* with an *`SQLState`* of 38000.

You can avoid having Derby wrap the exception if:

- The exception is an *SQLException*
- The range of the *SQLState* is 38001-38999

(This conforms to the SQL99 standard.)

Programming trigger actions

Derby allows you to create triggers. When you create a trigger, you define an action or set of actions that are executed when a database event occurs on a specified table. A *database event* is a delete, insert, or update operation.

For example, if you define a trigger for a delete on a particular table, the trigger action is executed whenever someone deletes a row or rows from the table.

The `CREATE TRIGGER` statement in the *Java DB Reference Manual* goes into detail of the complete `CREATE TRIGGER` syntax. This section provides information on defining the trigger action itself, which is only one aspect of creating triggers.

This section refers to the `CREATE TRIGGER` statement as the *trigger actions*.

Trigger action overview

A trigger action is a simple SQL statement.

For example:

```
CREATE TRIGGER . . .
DELETE FROM flightavailability
 WHERE flight_id IN (SELECT flight_id FROM flightavailability
  WHERE YEAR(flight_date) < 2005);)
```

A trigger action does have some limitations, though; for example, it cannot contain dynamic parameters or alter the table on which the trigger is defined. See "TriggerAction" in the *Java DB Reference Manual* for details.

Performing referential actions

Derby provides referential actions. Examples in this section are included to illustrate how to write triggers.

You can choose to use standard SQL referential integrity to obtain this functionality, rather than writing triggers. See the *Java DB Reference Manual* for more information on referential integrity.

Accessing before and after rows

Many trigger actions need to access the values of the rows being changed.

Such trigger actions need to know one or both of the following:

- the "before" values of the rows being changed (their values before the database event that caused the trigger to fire)
- the "after" values of the rows being changed (the values to which the database event is setting them)

Derby provides transition variables and transition tables for a trigger action to access these values. See "Referencing Old and New Values: The Referencing Clause" in the *Java DB Reference Manual*.

Examples of trigger actions

The following trigger action copies a row from the *flights* table into the *flight_history* table whenever any row gets inserted into *flights* and adds the comment "inserted from trig1" in the *status* column of the *flight_history* table.

```
CREATE TRIGGER trig1
AFTER UPDATE ON flights
REFERENCING OLD AS UPDATEDROW
FOR EACH ROW MODE DB2SQL
INSERT INTO flights_history
VALUES (UPDATEDROW.FLIGHT_ID, UPDATEDROW.SEGMENT_NUMBER,
UPDATEDROW.ORIG_AIRPORT, UPDATEDROW.DEPART_TIME,
UPDATED ROW.DEST_AIRPORT, UPDATEDROW.ARRIVE_TIME,
UPDATEDROW.MEAL, UPDATEDROW.FLYING_TIME, UPDATEDROW.MILES,
UPDATEDROW.AIRCRAFT,'INSERTED FROM trig1');
```

Triggers and exceptions

Exceptions raised by triggers have a statement severity; they roll back the statement that caused the trigger to fire.

This rule applies to nested triggers (triggers that are fired by other triggers). If a trigger action raises an exception (and it is not caught), the transaction on the current connection is rolled back to the point before the triggering event. For example, suppose Trigger A causes Trigger B to fire. If Trigger B throws an exception, the current connection is rolled back to the point before the statement in Trigger A that caused Trigger B to fire. Trigger A is then free to catch the exception thrown by Trigger B and continue with its work. If Trigger A does not throw an exception, the statement that caused Trigger A, as well as any work done in Trigger A, continues until the transaction in the current connection is either committed or rolled back. However, if Trigger A does not catch the exception from

Trigger B, it is as if Trigger A had thrown the exception. In that case, the statement that caused Trigger A to fire is rolled back, along with any work done by both of the triggers.

Aborting statements and transactions

You might want a trigger action to be able to abort the triggering statement or even the entire transaction.

Triggers that use the current connection are not permitted to commit or roll back the connection, so how do you do that? The answer is: have the trigger throw an exception, which is by default a statement-level exception (which rolls back the statement). The application-side code that contains the statement that caused the trigger to fire can then roll back the entire connection if desired. Programming triggers in this respect is no different from programming any database-side JDBC method.

Programming Derby-style table functions

Derby lets you create table functions. Table functions are functions which package up external data to look like Derby tables. The external data can be an XML file, a table in a foreign database, a live data feed--in short, any information source that can be presented as a JDBC *ResultSet*.

Derby-style table functions let you efficiently import foreign data into Derby tables. Table functions let you join Derby tables with any of the following data sources:

- XML-formatted reports and logs
- Queries that run in foreign databases
- Streaming data from sensors
- RSS feeds

See "CREATE FUNCTION statement" in the *Java DB Reference Manual* for the complete syntax needed to declare Derby-style table functions. The following topics provide information on how to write Java methods which wrap foreign data sources inside *ResultSet*s.

Overview of Derby-style table functions

A Derby-style table function is a method which returns a JDBC *ResultSet*.

Most of the *ResultSet* methods can be written as stubs which simply raise exceptions. However, the Derby-style table function must implement the following *ResultSet* methods:

- *next()*
- *close()*
- *wasNull()*
- *getXXX()* - When invoking a Derby-style table function at runtime, Derby calls a *getXXX()* method on each referenced column. The particular *getXXX()* method is based on the column's data type as declared in the `CREATE FUNCTION` statement. Preferred getXXX() methods for Derby-style table functions explains how Derby selects an appropriate *getXXX()* method. However, nothing prevents application code from calling other *getXXX()* methods on the *ResultSet*. The returned *ResultSet* needs to implement the *getXXX()* methods which Derby will call as well as all *getXXX()* methods which the application will call.

A Derby-style table function is materialized by a public static method which returns a *ResultSet*:

```
public static ResultSet read() {...}
```

The public static method is then bound to a Derby function name:

```
CREATE FUNCTION externalEmployees
()
RETURNS TABLE
(
  employeeId    INT,
  lastName      VARCHAR( 50 ),
  firstName     VARCHAR( 50 ),
  birthday      DATE
)
LANGUAGE JAVA
PARAMETER STYLE DERBY_JDBC_RESULT_SET
READS SQL DATA
EXTERNAL NAME 'com.acme.hrSchema.EmployeeTable.read'
```

To invoke a table function, wrap it in a TABLE constructor in the FROM list of a query. Note that the table alias (in this example "s") is a required part of the syntax:

```
INSERT INTO employees
  SELECT s.*
    FROM TABLE (externalEmployees() ) s;
```

Preferred *getXXX()* methods for Derby-style table functions

While scanning a Derby-style table function, Derby calls a preferred *getXXX()* method for each column, based on the column's data type. If Derby is running on a small device platform and presenting the JSR 169 interface to clients, then the methods which Derby calls are slightly different. This is because JSR 169 does not support BigDecimal.

The following table lists the preferred *getXXX()* method for each Derby data type.

Table 3. *getXXX()* Methods Called for Declared SQL Types

Column Type Declared by CREATE FUNCTION	*getXXX()* Method Called by Derby for JDBC 3.0 and 4.0	*getXXX()* Method Called by Derby for JSR 169
BIGINT	*getLong()*	Same
BLOB	*getBlob()*	Same
CHAR	*getString()*	Same
CHAR FOR BIT DATA	*getBytes()*	Same
CLOB	*getClob()*	Same
DATE	*getDate()*	Same
DECIMAL	*getBigDecimal()*	*getString()*
DOUBLE	*getDouble()*	Same
DOUBLE PRECISION	*getDouble()*	Same
FLOAT	*getDouble()*	Same
INTEGER	*getInt()*	Same
LONG VARCHAR	*getString()*	Same
LONG VARCHAR FOR BIT DATA	*getBytes()*	Same

Column Type Declared by CREATE FUNCTION	***getXXX()* Method Called by Derby for JDBC 3.0 and 4.0**	***getXXX()* Method Called by Derby for JSR 169**
NUMERIC	*getBigDecimal()*	*getString()*
REAL	*getFloat()*	Same
SMALLINT	*getShort()*	Same
TIME	*getTime()*	Same
TIMESTAMP	*getTimestamp()*	Same
VARCHAR	*getString()*	Same
VARCHAR FOR BIT DATA	*getBytes()*	Same
XML	**Not supported**	**Not supported**

Example Derby-style table function

The following simple table function selects rows from a foreign database.

```
package com.acme.hrSchema;

import java.sql.*;

/**
 * Sample Table Function for reading the employee table in an
 * external database.
 */
public class EmployeeTable
{
    public  static  ResultSet   read()
        throws SQLException
    {
        Connection          conn = getConnection();
        PreparedStatement   ps = conn.prepareStatement( "select * from
 hrSchema.EmployeeTable" );

        return ps.executeQuery();
    }

    protected  static  Connection    getConnection()
        throws SQLException
    {
        String  EXTERNAL_DRIVER = "com.mysql.jdbc.Driver";

        try { Class.forName( EXTERNAL_DRIVER ); }
        catch (ClassNotFoundException e) { throw new SQLException( "Could
 not find class " + EXTERNAL_DRIVER ); }

        Connection          conn = DriverManager.getConnection
            ( "jdbc:mysql://localhost/hr?user=root&password=mysql-passwd"
 );

        return conn;
    }
}
```

Optimizer support for Derby-style table functions

This topic explains how to fine-tune the Derby optimizer's decision about where to place a table function in the join order.

By default, the Derby optimizer makes the following assumptions about a table function:

- **Expensive** - It is expensive to create and loop through the rows of the table function. This makes it likely that the optimizer will place the table function in an outer slot of the join order so that it will not be looped through often.
- **Repeatable** - The table function can be instantiated multiple times with the same results. This is probably true for most table functions. However, some table functions may open read-once streams. If the optimizer knows that a table function is repeatable, then the optimizer can place the table function in an inner slot where the function can be invoked multiple times. If a table function is not repeatable, then the optimizer must either place it in the outermost slot or invoke the function once and store its contents in a temporary table.

The user can override this optimizer behavior by giving the optimizer more information. Here's how to do this:

- **No-arg constructor** - The table function's class must have a public constructor whose signature has no arguments.
- **VTICosting** - The class must also implement *org.apache.derby.vti.VTICosting*. This involves implementing the following methods as described in Measuring the cost of Derby-style table functions and Example VTICosting implementation:
  - *getEstimatedCostPerInstantiation()* - This method estimates the cost of invoking the table function and looping through its rows. The returned value adds together two estimates:
    - Empty table - This is the cost of invoking the table function, even if it contains 0 rows. See the description of variable **E** in Measuring the cost of Derby-style table functions.
    - Scanning - This is the cost of looping through all of the rows returned by the table function. See the calculation of **P*N** in Measuring the cost of Derby-style table functions.
  - *getEstimatedRowCount()* - This guesses the number of rows returned by invoking the table function.
  - *supportsMultipleInstantiations()* - This returns false if the table function returns different results when invoked more than once.

For more information, see:

Measuring the cost of Derby-style table functions

This topic shows how to measure the cost of a Derby-style table function.

The following formula describes how to estimate the value returned by *VTICosting.getEstimatedCostPerInstantiation()*:

C = I * A

where

- **C** = The estimated **C**ost for creating and running the table function. That is, the value returned by *VTICosting.getEstimatedCostPerInstantiation()*. In general, **C**ost is a measure of time in milliseconds.
- **I** = The optimizer's **I**mprecision. A measure of how skewed the optimizer's estimates tend to be in your particular environment. See below for instructions on how to estimate this **I**mprecision.
- **A** = The **A**ctual time in milliseconds which it takes to create and run this table function.

Calculating the optimizer's imprecision

We treat optimizer **I**mprecision as a constant across the runtime environment. The following formula describes it:

I = O / T

where

- **O** = The **O**ptimizer's estimated cost for a plan.
- **T** = The **T**otal runtime in milliseconds for the plan.

To estimate these values, turn on Derby statistics collection and run the following experiment several times, averaging the results:

- **Select** = Select all of the rows from a big table.
- **Record** = In the statistics output, look for the ResultSet which represents the table scan. That scan has a field labelled "optimizer estimated cost". That's **O**. Now look for the fields in that ResultSet's statistics labelled "constructor time", "open time", "next time", and "close time". Add up all of those fields. That total is **T**.

For example:

```
MAXIMUMDISPLAYWIDTH 7000;

CALL SYSCS_UTIL.SYSCS_SET_RUNTIMESTATISTICS(1);
CALL SYSCS_UTIL.SYSCS_SET_STATISTICS_TIMING(1);

select * from T;

values SYSCS_UTIL.SYSCS_GET_RUNTIMESTATISTICS();
```

Calculating the actual runtime cost of a table function

The following formula explains how to compute the **A**ctual runtime cost for the table function:

A = (P * N) + E

where

- **P** = The runtime spent **P**er row (in milliseconds).
- **N** = The **N**umber of rows in the table function.
- **E** = The time spent creating an **E**mpty instance of the table function which has no rows in it. Usually, **P * N** dwarfs **E**. That is, the table function instantiation cost is very small compared to the actual cost of looping through the rows. However, for some table functions, **E** may be significant and may dominate the table function's cost when **N** is small.

You may know that **E** is basically 0. If so, you can skip this step. Otherwise, to estimate **E**, turn on Derby statistics collection and run the following experiment several times, averaging the results:

- **Short-circuit** = Short-circuit the next() method of the *ResultSet* returned by your Derby-style Table Function so that it returns *false* the first time it is called. This makes it appear that the *ResultSet* has no rows.
- **Select** = Select all of the rows from the table function.
- **Record** = In the statistics output, look for the VTIResultSet which represents the table function scan. Add up the values of the fields in that VTIResultSet's statistics labelled "constructor time", "open time", "next time", and "close time". That total is **E**.

To estimate **P**, turn on Derby statistics collection and run the following experiment several times, averaging the results:

- **Select** = Select all of the rows from the table function.

- **Record** = In the statistics output, look for the VTIResultSet which represents the table function scan. Add up the values of the fields in that VTIResultSet's statistics labelled "constructor time", "open time", "next time", and "close time". Subtract **E** from the result. Now divide by the value of the field "Rows seen". The result is **P**.

Computing the value returned by *getEstimatedCostPerInstantiation()*

Putting all of this together, the following formula describes the value returned by your table function's *VTICosting.getEstimatedCostPerInstantiation()* method.

C = O/T * [(P * N) + E]

Example VTICosting implementation

Once you have measured your table function's cost, you can write the *VTICosting* methods.

Optimizer fine-tuning can be added to the *EmployeeTable* table function shown before in Example Derby-style table function:

```
package com.acme.hrSchema;

import java.io.Serializable;
import java.sql.*;

import org.apache.derby.vti.VTICosting;
import org.apache.derby.vti.VTIEnvironment;

/**
 * Tuned table function.
 */
public class TunedEmployeeTable extends EmployeeTable implements
 VTICosting
{
    public TunedEmployeeTable() {}

    public double getEstimatedRowCount( VTIEnvironment optimizerState )
 throws SQLException
    {
        return getRowCount( optimizerState );
    }

    public double getEstimatedCostPerInstantiation( VTIEnvironment
 optimizerState ) throws SQLException
    {
        double      I = 100.0;  // optimizer imprecision
        double      P = 10.0;   // cost per row in milliseconds
        double      E = 0.0;    // cost of instantiating the external
 ResultSet
        double      N = getRowCount( optimizerState );

        return I * ( ( P * N ) + E );
    }

    public boolean supportsMultipleInstantiations( VTIEnvironment
 optimizerState ) throws SQLException
    {
        return true;
    }


 ////////////////////////////////////////////////////////////////////////
//////

    private double  getRowCount( VTIEnvironment optimizerState )
        throws SQLException
    {
        String              ROW_COUNT_KEY = "rowCountKey";
```

```
        Double          estimatedRowCount = (Double) getSharedState(
 optimizerState, ROW_COUNT_KEY );

        if ( estimatedRowCount == null )
        {
            Connection                  conn = getConnection();
            PreparedStatement       ps = conn.prepareStatement( "select
 count(*) from hrSchema.EmployeeTable" );
            ResultSet                      rs = ps.executeQuery();

            rs.next();
            estimatedRowCount = new Double( rs.getDouble( 1 ) );

            setSharedState( optimizerState, ROW_COUNT_KEY,
 estimatedRowCount );

            rs.close();
            ps.close();
            conn.close();
        }

        return estimatedRowCount.doubleValue();
    }

    private Serializable  getSharedState( VTIEnvironment optimizerState,
 String key )
    { return (Serializable) optimizerState.getSharedState( key ); }
    private void    setSharedState( VTIEnvironment optimizerState, String
 key, Serializable value )
    { optimizerState.setSharedState( key, value ); }
}
```

Controlling Derby application behavior

This section looks at some advanced Derby application concepts.

The JDBC connection and transaction model

Session and transaction capabilities for SQL are handled through JDBC routines, not by SQL commands.

JDBC defines a system session and transaction model for database access. A *session* is the duration of one connection to the database and is handled by a JDBC *Connection* object.

Connections

A *Connection* object represents a connection with a database.

Within the scope of one *Connection*, you access only a single Derby database. (Database-side JDBC routines can allow you to access more than one database in some circumstances.) A single application might allow one or more *Connections* to Derby, either to a single database or to many different databases, provided that all the databases are within the same system.

With DriverManager, you use the connection URL as an argument to get the *getConnection* method to specify which database to connect to and other details.

The following example shows an application establishing three separate connections to two different databases in the current system.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sample");
System.out.println("Connected to database sample");
conn.setAutoCommit(false);
Connection conn2 = DriverManager.getConnection(
    "jdbc:derby:newDB;create=true");
System.out.println("Created AND connected to newDB");
conn2.setAutoCommit(false);
Connection conn3 = DriverManager.getConnection(
    "jdbc:derby:newDB");
System.out.println("Got second connection to newDB");
conn3.setAutoCommit(false);
```

A *Connection* object has no association with any specific thread; during its lifetime, any number of threads might have access to it, as controlled by the application.

Statements

To execute SQL statements against a database, an application uses *Statements* (`java.sql.Statement`) and *PreparedStatements* (`java.sql.PreparedStatement`), or *CallableStatements* (`java.sql.CallableStatement`) for stored procedures.

Because *PreparedStatement* extends *Statement* and *CallableStatement* extends *PreparedStatement*, this section refers to both as *Statements*. *Statements* are obtained from and are associated with a particular *Connection*.

ResultSets and Cursors

Executing a *Statement* that returns values gives a `ResultSet` (`java.sql.ResultSet`), allowing the application to obtain the results of the statement.

Only one *ResultSet* can be open for a particular *Statement* at any time, as per the JDBC specification.

Thus, executing a *Statement* automatically closes any open *ResultSet* generated by an earlier execution of that *Statement*.

For this reason, you must use a different *Statement* to update a cursor (a named *ResultSet*) from the one used to generate the cursor.

The names of open cursors must be unique within a *Connection*.

Nested connections

SQL statements can include routine invocations. If these routines interact with the database, they must use a *Connection*.

Transactions

A *transaction* is a set of one or more SQL statements that make up a logical unit of work that you can either commit or roll back and that will be recovered in the event of a system failure.

All the statements in the transaction are *atomic*. A transaction is associated with a single *Connection* object (and database). A transaction cannot span *Connections* (or databases).

Derby permits schema and data manipulation statements (DML) to be intermixed within a single transaction. If you create a table in one transaction, you can also insert into it in that same transaction. A schema manipulation statement (DDL) is not automatically committed when it is performed, but participates in the transaction within which it is issued. Because DDL requires exclusive locks on system tables, keep transactions that involve DDL short.

Transactions when auto-commit is disabled

When auto-commit is disabled, you use a *Connection* object's *commit* and *rollback* methods to commit or roll back a transaction.

The *commit* method makes permanent the changes resulting from the transaction and releases locks. The *rollback* method undoes all the changes resulting from the transaction and releases locks. A transaction encompasses all the SQL statements executed against a single *Connection* object since the last *commit* or *rollback*.

You do not need to explicitly begin a transaction. You implicitly end one transaction and begin a new one after disabling auto-commit, changing the isolation level, or after calling *commit* or *rollback*.

Committing a transaction also closes all *ResultSet* objects excluding the *ResultSet* objects associated with cursors with holdability `true`. The default holdability of the cursors is `true` and *ResultSet* objects associated with them need to be closed explicitly. A commit will not close such *ResultSet* objects. It also releases any database locks currently held by the *Connection*, whether or not these objects were created in different threads.

Using auto-commit

A new connection to a Derby database is in auto-commit mode by default, as specified by the JDBC standard.

Auto-commit mode means that when a statement is completed, the method *commit* is called on that statement automatically. Auto-commit in effect makes every SQL statement a transaction. The commit occurs when the statement completes or the next statement is executed, whichever comes first. In the case of a statement returning a *ResultSet*,

the statement completes when the last row of the `ResultSet` has been retrieved or the `ResultSet` has been closed explicitly.

Some applications might prefer to work with Derby in auto-commit mode; some might prefer to work with auto-commit turned off. You should be aware of the implications of using either model.

You should be aware of the following when you use auto-commit:

- *Cursors*

  You cannot use auto-commit if you do any positioned updates or deletes (that is, an update or delete statement with a WHERE CURRENT OF clause) on cursors which have the `ResultSet.CLOSE_CURSORS_AT_COMMIT` holdability value set.

  Auto-commit automatically closes cursors that are explicitly opened with the `ResultSet.CLOSE_CURSORS_AT_COMMIT` value, when you do any in-place updates or deletes.

  An updatable cursor declared to be held across commit (this is the default value) can execute updates and issue multiple commits before closing the cursor. After an explicit or implicit commit, a holdable forward-only cursor must be repositioned with a call to the `next` method before it can accessed again. In this state, the only other valid operation besides calling `next` is calling `close`.
- *Database-side JDBC routines (routines using nested connections)*

  You cannot execute functions within SQL statements if those functions perform a commit or rollback on the current connection. Since in auto-commit mode all SQL statements are implicitly committed, Derby turns off auto-commit during execution of database-side routines and turns it back on when the statement completes.

  Routines that use nested connections are not permitted to turn auto-commit on or off.
- *Table-level locking and the SERIALIZABLE isolation level*

  When an application uses table-level locking and the SERIALIZABLE isolation level, all statements that access tables hold at least shared table locks. Shared locks prevent other transactions that update data from accessing the table. A transaction holds a lock on a table until the transaction commits. *So even a SELECT statement holds a shared lock on a table until its connection commits and a new transaction begins.*

Table 4. Summary of Application Behavior with Auto-Commit On or Off

Topic	Auto-Commit On	Auto-Commit Off
Transactions	Each statement is a separate transaction.	Commit() or rollback() completes a transaction.
Database-side JDBC routines (routines that use nested connections)	Auto-commit is turned off.	Works (no explicit commits or rollbacks are allowed).
Updatable cursors	Works for holdable cursors; does not work for non-holdable cursors.	Works.
	Works.	

Topic	Auto-Commit On	Auto-Commit Off
Multiple connections accessing the same data		Works. Lower concurrency when applications use SERIALIZABLE isolation mode and table-level locking.
Updatable ResultSets	Works.	Works.
Savepoints	Does not work.	Works.

Turning off auto-commit

You can disable auto-commit with the *Connection* class's *setAutoCommit* method.

```
conn.setAutoCommit(false);
```

Explicitly closing Statements, ResultSets, and Connections

You should explicitly close *Statements*, *`ResultSets,`* and *`Connections`* when you no longer need them.

Connections to Derby are resources external to an application, and the garbage collector will not close them automatically.

For example, close a *Statement* object using its *`close`* method; close a *`Connection`* object using its *`close`* method. If auto-commit is disabled, active transactions need to be explicitly committed or rolled back before closing the connection

Statement versus transaction runtime rollback

When an SQL statement generates an exception, this exception results in a *runtime rollback*. A runtime rollback is a system-generated rollback of a statement or transaction by Derby, as opposed to an explicit *`rollback`* call from your application.

Extremely severe exceptions, such as disk-full errors, shut down the system, and the transaction is rolled back when the database is next booted. Severe exceptions, such as deadlock, cause transaction rollback; Derby rolls back all changes since the beginning of the transaction and implicitly begins a new transaction. Less severe exceptions, such as syntax errors, result in statement rollback; Derby rolls back only changes made by the statement that caused the error. The application developer can insert code to explicitly roll back the entire transaction if desired.

Derby supports partial rollback through the use of savepoints. See Using savepoints for more information.

Using savepoints

The *Savepoint* interface contains methods to set, release, or roll back a transaction to designated savepoints. Once a savepoint has been set, the transaction can be rolled back to that savepoint without affecting preceding work. Savepoints provide finer-grained control of transactions by marking intermediate points within a transaction.

Setting and rolling back to a savepoint

The *Connection.setSavepoint* method sets a savepoint within the current transaction. The *Connection.rollback* method is overloaded to take a savepoint argument.

The code example below inserts a row into a table, sets the savepoint `svpt1`, and then inserts a second row. When the transaction is later rolled back to `svpt1`, the second insertion is undone, but the first insertion remains intact. In other words, when the transaction is committed, only the row containing '1' will be added to TABLE1.

```
conn.setAutoCommit(false); // Autocommit must be off to use savepoints.
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES(1)");
// set savepoint
Savepoint svpt1 = conn.setSavepoint("S1");
rows = stmt.executeUpdate("INSERT INTO TABLE1 (COL1) VALUES (2)");
...
conn.rollback(svpt1);
...
conn.commit();
```

Releasing a savepoint

The method *Connection.releaseSavepoint* takes a *Savepoint* object as a parameter and removes it from the current transaction. Once a savepoint has been released, attempting to reference it in a rollback operation will cause an *SQLException* to be thrown.

Any savepoints that have been created in a transaction are automatically released and become invalid when the transaction is committed or when the entire transaction is rolled back.

Rolling a transaction back to a savepoint automatically releases and makes invalid any other savepoints created after the savepoint in question.

Rules for savepoints

The savepoint cannot be set within a batch of statements to enable partial recovery. If a savepoint is set any time before the method *executeBatch* is called, it is set before any of the statements that have been added to the batch are executed.

A savepoint can be reused after it has been released explicitly (by issuing a release of the savepoint) or implicitly (by issuing a connection commit/rollback to that savepoint or to a savepoint declared earlier than that savepoint).

It is possible to nest savepoints, but only in an embedded environment.

Result set and cursor mechanisms

A result set maintains a cursor, which points to its current row of data. It can be used to step through and process the rows one by one.

In Derby, any SELECT statement generates a cursor which can be controlled by a *java.sql.ResultSet* object. Derby does not support SQL-92's DECLARE CURSOR language construct to create cursors, however Derby supports positioned deletes and positioned updates of updatable cursors.

Simple non-updatable result sets

This example is an excerpt from a sample JDBC application that generates a result set with a simple SELECT statement and then processes the rows.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:sample");
Statement s = conn.createStatement();
s.execute("set schema 'SAMP'");
//note that autocommit is on--it is on by default in JDBC
ResultSet rs = s.executeQuery(
    "SELECT empno, firstnme, lastname, salary, bonus, comm "
     + "FROM samp.employee");
/** a standard JDBC ResultSet. It maintains a
  *  cursor that points to the current row of data. The cursor
  *  moves down one row each time the method next() is called.
  *  You can scroll one way only--forward--with the next()
  *  method. When auto-commit is on, after you reach the
```

```
  *  last row the statement is considered completed
  *  and the transaction is committed.
  */
System.out.println( "last name" + "," + "first name" + ": earnings");
/* here we are scrolling through the result set
with the next() method.*/
while (rs.next()) {
    // processing the rows
    String firstnme = rs.getString("FIRSTNME");
    String lastName = rs.getString("LASTNAME");
    BigDecimal salary = rs.getBigDecimal("SALARY");
    BigDecimal bonus = rs.getBigDecimal("BONUS");
    BigDecimal comm = rs.getBigDecimal("COMM");
    System.out.println( lastName + ", " + firstnme + ": "
                       + (salary.add(bonus.add(comm))));
}
rs.close();
// once we've iterated through the last row,
// the transaction commits automatically and releases
//shared locks
s.close();
```

Updatable result sets

Updatable result sets in Derby can be updated by using result set update methods (`updateRow()`,`deleteRow()` and `insertRow()`), or by using positioned update or delete queries.

Both scrollable and non-scrollable result sets can be updatable in Derby.

If the query which was executed to create the result set is not updatable, Derby will downgrade the concurrency mode to `ResultSet.CONCUR_READ_ONLY`, and add a warning about this on the `ResultSet`. The compilation of the query fails if the result set cannot be updatable, and contains a `FOR UPDATE` clause.

Positioned updates and deletes can be performed if the query contains `FOR UPDATE` or if the concurrency mode for the result set is `ResultSet.CONCUR_UPDATABLE`.

To use the result set update methods, the concurrency mode for the result set must be `ResultSet.CONCUR_UPDATABLE`. The query does not need to contain `FOR UPDATE` to use these methods.

Updatable cursors lock the current row with an update lock when positioned on the row, regardless of isolation level. Therefore, to avoid excessive locking of rows, only use concurrency mode `ResultSet.CONCUR_UPDATABLE` or the `FOR UPDATE` clause when you actually need to update the rows. For more information about locking, see Types and scope of locks in Derby systems.

Requirements for updatable result sets

Only specific SELECT statements- simple accesses of a single table-allow you to update or delete rows as you step through them.

For more information, see "SELECT statement" and "FOR UPDATE clause" in the *Java DB Reference Manual*.

Forward only updatable result sets

A forward only updatable result set maintains a cursor which can only move in one direction (forward), and also update rows.

To create a forward only updatable result set, the statement has to be created with concurrency mode `ResultSet.CONCUR_UPDATABLE` and type `ResultSet.TYPE_FORWARD_ONLY`.

Note: The default type is `ResultSet.TYPE_FORWARD_ONLY`.

Example of using `ResultSet.updateXXX() + ResultSet.updateRow()` to update a row:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
  "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
  "FROM EMPLOYEE");

while (uprs.next()) {
    int newBonus = uprs.getInt("BONUS") + 100;
    uprs.updateInt("BONUS", newBonus);
    uprs.updateRow();
}
```

Example of using `ResultSet.deleteRow()` to delete a row:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
  "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
  "FROM EMPLOYEE");

while (uprs.next()) {
    if (uprs.getInt("WORKDEPT")==300) {
       uprs.deleteRow();
    }
}
```

Visibility of changes

- After an update or delete is made on a forward only result set, the result set's cursor is no longer on the row just updated or deleted, but immediately before the next row in the result set (it is necessary to move to the next row before any further row operations are allowed). This means that changes made by `ResultSet.updateRow()` and `ResultSet.deleteRow()` are never visible.
- If a row has been inserted, i.e using `ResultSet.insertRow()` it may be visible in a forward only result set.

Conflicting operations

The current row of the result set cannot be changed by other transactions, since it will be locked with an update lock. Result sets held open after a commit have to move to the next row before allowing any operations on it.

Some conflicts may prevent the result set from doing updates/deletes:
- If the current row is deleted by a statement in the same transaction, calls to `ResultSet.updateRow()` will cause an exception, since the cursor is no longer positioned on a valid row.

Scrollable updatable result sets

A scrollable updatable result set maintains a cursor which can both scroll and update rows.

Derby only supports scrollable insensitive result sets. To create a scrollable insensitive result set which is updatable, the statement has to be created with concurrency mode `ResultSet.CONCUR_UPDATABLE` and type `ResultSet.TYPE_SCROLL_INSENSITIVE`.

Example of using result set update methods to update a row:

```
 Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
```

```
                                        ResultSet.CONCUR_UPDATABLE);
 ResultSet uprs = stmt.executeQuery(
   "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
   "FROM EMPLOYEE");

 uprs.absolute(5); // update the fifth row
 int newBonus = uprs.getInt("BONUS") + 100;
 uprs.updateInt("BONUS", newBonus);
 uprs.updateRow();
```

Example of using `ResultSet.deleteRow()` to delete a row:

```
 Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                        ResultSet.CONCUR_UPDATABLE);
 ResultSet uprs = stmt.executeQuery(
   "SELECT FIRSTNAME, LASTNAME, WORKDEPT, BONUS " +
   "FROM EMPLOYEE");

 uprs.last();
 uprs.relative(-5); // moves to the 5th from the last row
 uprs.deleteRow();
```

Visibility of changes

- Changes caused by other statements, triggers and other transactions (others) are considered as other changes, and are not visible in scrollable insensitive result sets.
- Own updates and deletes are visible in Derby's scrollable insensitive result sets. **Note:** Derby handles changes made using positioned updates and deletes as own changes, so when made via a result set's cursor such changes are also visible in that result set.
- Rows inserted to the table may become visible in the result set.
- `ResultSet.rowDeleted()` returns true if the row has been deleted using the cursor or result set. It does not detect deletes made by other statements or transactions. Note that the method will also work for result sets with concurrency CONCUR_READ_ONLY if the underlying result set is FOR UPDATE and a cursor was used to delete the row.
- `ResultSet.rowUpdated()` returns true if the row has been updated using the cursor or result set. It does not detect updates made by other statements or transactions. Note that the method will also work for result sets with concurrency CONCUR_READ_ONLY if the underlying result set is FOR UPDATE and a cursor was used to update the row.
- **Note:** Both `ResultSet.rowUpdated()` and `ResultSet.rowDeleted()` return true if the row first is updated and later deleted.

Please be aware that even if changes caused by others are not visible in the result set, SQL operations, including positioned updates, which access the current row will read and use the row data as it is in the database, not as it is reflected in the result set.

Conflicting operations

A conflict may occur in scrollable insensitive result sets if a row is updated/deleted by another committed transaction, or if a row is updated by another statement in the same transaction. The row which the cursor is positioned on is locked, however once it moves to another row, the lock may be released depending on transaction isolation level. This means that rows in the scrollable insensitive result set may have been updated/deleted by other transactions after they were fetched.

Since the result set is **insensitive**, it will not detect the changes made by others. When doing updates using the result set, conflicting changes on the columns being changed will be overwritten.

Some conflicts may prevent the result set from doing updates/deletes:

- The row has been deleted after it was read into the result set: Scrollable insensitive result sets will give a warning with *SQLState* 01001 .
- The table has been compressed: Scrollable insensitive result sets will give a warning with *SQLState* 01001. A compress conflict may happen if the cursor is held over a commit. This is because the table intent lock is released on commit, and not reclaimed until the cursor moves to another row.

To avoid conflicts with other transactions, you may increase the transaction isolation level to repeatable read or serializable. This will make the transaction hold locks on the rows which have been read until it commits.

Note: When you use holdable result sets, be aware that the locks will be released on commit, and conflicts may occur regardless of isolation level. You should probably avoid using holdable result sets if your application relies on transactional behavior for the result set.

Inserting rows with updatable result sets

Updatable result set can be used to insert rows to the table, by using `ResultSet.insertRow()`.

When inserting a row, each column in the insert row that does not allow null as a value and does not have a default value must be given a value using the appropriate update method. If the inserted row satisfies the query predicate, it may become visible in the result set.

Example of using `ResultSet.insertRow()` to insert a row:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE);
ResultSet uprs = stmt.executeQuery(
  "SELECT firstname, lastname, workdept, bonus " +
  "FROM employee");
uprs.moveToInsertRow();
uprs.updateString("FIRSTNAME", "Andreas");
uprs.updateString("LASTNAME", "Korneliussen");
uprs.updateInt("WORKDEPT", 123);
uprs.insertRow();
uprs.moveToCurrentRow();
```

Naming or accessing the name of a cursor

There is no SQL language command to *assign* a name to a cursor. You can use the JDBC *setCursorName* method to assign a name to a *ResultSet* that allows positioned updates and deletes.

You assign a name to a *ResultSet* with the *setCursorName* method of the *Statement* interface. You assign the name to a cursor before executing the *Statement* that will generate it.

```
Statement s3 = conn.createStatement();
// name the statement so we can reference the result set
// it generates
s3.setCursorName("UPDATABLESTATEMENT");
// we will be able to use the following statement later
// to access the current row of the cursor
// a result set needs to be obtained prior to using the
// WHERE CURRENT syntax
ResultSet rs = s3.executeQuery("select * from
    FlightBookings FOR UPDATE of number_seats");
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE FlightBookings SET number_seats = ? " +
       "WHERE CURRENT OF UPDATABLESTATEMENT");
```

Typically, you do not assign a name to the cursor, but let the system generate one for you automatically. You can determine the system-generated cursor name of a *ResultSet*

generated by a SELECT statement using the *ResultSet* class's *getCursorName* method.

```
PreparedStatement ps2 = conn.prepareStatement(
    "UPDATE employee SET bonus = ? WHERE CURRENT OF "+
        Updatable.getCursorName());
```

Extended updatable result set example

```
Connection conn = DriverManager.getConnection("jdbc:derby:sample");
conn.setAutoCommit(false);

// Create the statement with concurrency mode CONCUR_UPDATABLE
// to allow result sets to be updatable
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                                      ResultSet.CONCUR_UPDATABLE,
                                      ResultSet.CLOSE_CURSORS_AT_COMMIT);
// Updatable statements have some requirements
// for example, select must be on a single table
ResultSet uprs = stmt.executeQuery(
    "SELECT FIRSTNME, LASTNAME, WORKDEPT, BONUS " +
    "FROM EMPLOYEE FOR UPDATE of BONUS"); // Only bonus can be updated

String theDept="E21";

while (uprs.next()) {
    String firstnme = uprs.getString("FIRSTNME");
    String lastName = uprs.getString("LASTNAME");
    String workDept = uprs.getString("WORKDEPT");
    BigDecimal bonus = uprs.getBigDecimal("BONUS");
    if (workDept.equals(theDept)) {
        // if the current row meets our criteria,
        // update the updatable column in the row
        uprs.updateBigDecimal("BONUS",
 bonus.add(BigDecimal.valueOf(250L)));
        uprs.updateRow();
        System.out.println("Updating bonus for employee:" +
                           firstnme + lastName);
    }
}
conn.commit(); // commit the transaction
// close object
uprs.close();
stmt.close();
// Close connection if the application does not need it any more
conn.close();
```

Result sets and auto-commit

Except for the result sets associated with holdable cursors, issuing a commit will cause all result sets on your connection to be closed.

The JDBC application is not required to have auto-commit off when using update methods on updatable result set, even if the result set is not holdable. Positioned updates and deletes cannot be used in combination with autocommit and non-holdable result sets.

Scrollable result sets

JDBC provides two types of result sets that allow you to scroll in either direction or to move the cursor to a particular row. Derby supports one of these types: scrollable insensitive result sets (`ResultSet.TYPE_SCROLL_INSENSITIVE`).

When you use a result set of type of type `ResultSet.TYPE_SCROLL_INSENSITIVE`, Derby materializes rows from the first one in the result set up to the one with the biggest row number as the rows are requested. The materialized rows will be backed to disk if necessary, to avoid excessive memory usage.

Insensitive result sets, in contrast to sensitive result sets, cannot see changes made by others on the rows which have been materialized. Derby allows updates of scrollable insensitive result sets; see Visibility of changes, which also explains visibility of own changes.

Note: Derby does not support result sets of type `ResultSet.TYPE_SCROLL_SENSITIVE`.

```
//autocommit does not have to be off because even if
//we accidentally scroll past the last row, the implicit commit
//on the the statement will not close the result set because result sets
//are held over commit by default
conn.setAutoCommit(false);
Statement s4 = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_READ_ONLY);
s4.execute("set schema 'SAMP'");
ResultSet scroller=s4.executeQuery(
    "SELECT sales_person, region, sales FROM sales " +
    "WHERE sales > 8 ORDER BY sales DESC");
if (scroller.first()) { // One row is now materialized
    System.out.println("The sales rep who sold the highest number" +
                       " of sales is " +
                       scroller.getString("SALES_PERSON"));
} else {
    System.out.println("There are no rows.");
}
scroller.beforeFirst();
scroller.afterLast();   // By calling afterlast(), all rows will be
 materialized
scroller.absolute(3);
if (!scroller.isAfterLast()) {
    System.out.println("The employee with the third highest number " +
                       "of sales is " +
                       scroller.getString("SALES_PERSON") + ", with " +
                       scroller.getInt("SALES") + " sales");
}
if (scroller.isLast()) {
    System.out.println("There are only three rows.");
}
if (scroller.last()) {
    System.out.println("The least highest number " +
                       "of sales of the top three sales is: " +
                       scroller.getInt("SALES"));
}
scroller.close();
s4.close();
conn.commit()
conn.close();
System.out.println("Closed connection");
```

Holdable result sets

The holdable result set feature permits an application to keep result sets open after implicit or explicit commits. By default, the cursor controlled by the result set is held open after a commit.

Note: Derby also supports non-holdable result sets.

When you create a statement, you can specify that the result set will be automatically closed when a commit occurs. Result sets are automatically closed when a transaction aborts, whether or not they have been specified to be held open.

To specify whether a result set should be held open after a commit takes place, supply one of the following *ResultSet* parameters to the `Connection` method `createStatement`, `prepareStatement`, or `prepareCall`:

- *CLOSE_CURSORS_AT_COMMIT*

  Result sets are closed when an implicit or explicit commit is performed.
- *HOLD_CURSORS_OVER_COMMIT*

  Result sets are held open when a commit is performed, implicitly or explicitly. This is the default behavior.

The method *Statement.getResultSetHoldability()* indicates whether a result set generated by the Statement object stays open or closes, upon commit. See the *Java DB Reference Manual* for more information.

When an implicit or explicit commit occurs, result sets that hold cursors open behave as follows:

- Open result sets remain open. Non-scrollable result sets becomes positioned before the next logical row of the result set. Scrollable insensitive result sets keep their current position.
- When the session is terminated, the result set is closed and destroyed.
- All locks are released, including locks protecting the current cursor position.
- For non-scrollable result sets, immediately following a commit, the only valid operations that can be performed on the *ResultSet* object are:
  - positioning the result set to the next row with `ResultSet.next()`.
  - closing the result set with `ResultSet.close()`.

When a rollback or rollback to savepoint occurs, either explicitly or implicitly, the following behavior applies:
- All open result sets are closed.
- All locks acquired during the unit of work are released.

Note: Holdable result sets do not work with XA transactions in Derby. When working with XA transactions, the result set should be opened with holdability `ResultSet.CLOSE_CURSORS_AT_COMMIT`.

Holdable result sets and autocommit

When autocommit is on, a positioned update or delete statement will automatically cause the transaction to commit.

If the result set has holdability `ResultSet.CLOSE_CURSORS_AT_COMMIT`, combined with autocommit on, Derby gives an exception on positioned updates and deletes because the cursor is closed immediately before the positioned statement is commenced, as mandated by JDBC. In contrast, no such implicit commit is done when using result set updates methods.

Non-holdable result set example

The following example uses *Connection.createStatement* to return a `ResultSet` that will close after a commit is performed.

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt =
conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                  ResultSet.CONCUR_READ_ONLY,
                  ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

Locking, concurrency, and isolation

This section discusses topics pertinent to multi-user systems, in which concurrency is important.

Derby is configured by default to work well for multi-user systems. For single-user systems, you might want to tune your system so that it uses fewer resources; see Lock granularity.

Isolation levels and concurrency

Derby provides four transaction isolation levels. Setting the transaction isolation level for a connection allows a user to specify how severely the user's transaction should be isolated from other transactions.

For example, it allows you to specify whether transaction A is allowed to make changes to data that have been viewed by transaction B before transaction B has committed.

A connection determines its own isolation level, so JDBC provides an application with a way to specify a level of transaction isolation. It specifies four levels of transaction isolation. The higher the transaction isolation, the more care is taken to avoid conflicts; avoiding conflicts sometimes means locking out transactions. Lower isolation levels thus allow greater concurrency.

Inserts, updates, and deletes always behave the same no matter what the isolation level is. Only the behavior of select statements varies.

To set isolation levels you can use the JDBC *Connection.setTransactionIsolation* method or the SQL SET ISOLATION statement.

If there is an active transaction, the network client driver always commits the active transaction, whether you use the JDBC *Connection.setTransactionIsolation* method or the SQL SET ISOLATION statement. It does this even if the method call or statement does not actually change the isolation level (that is, if it sets the isolation level to its current value). The embedded driver also always commits the active transaction if you use the SET ISOLATION statement. However, if you use the *Connection.setTransactionIsolation* method, the embedded driver commits the active transaction only if the call to *Connection.setTransactionIsolation* actually changes the isolation level.

The names of the isolation levels are different, depending on whether you use a JDBC method or SQL statement. Mapping of JDBC transaction isolation levels to Derby isolation levels shows the equivalent names for isolation levels whether they are set through the JDBC method or an SQL statement.

Table 5. Mapping of JDBC transaction isolation levels to Derby isolation levels

Isolation levels for JDBC	Isolation levels for SQL
Connection.TRANSACTION_READ_UNCOMMITT (ANSI level 0)	UR, DIRTY READ, READ UNCOMMITTED
Connection.TRANSACTION_READ_COMMITTED (ANSI level 1)	CS, CURSOR STABILITY, READ COMMITTED
Connection.TRANSACTION_REPEATABLE_REAI (ANSI level 2)	RS
Connection.TRANSACTION_SERIALIZABLE (ANSI level 3)	RR, REPEATABLE READ, SERIALIZABLE

These levels allow you to avoid particular kinds of transaction anomalies, which are described in Transaction Anomalies.

Table 6. Transaction Anomalies

Anomaly	Example
Dirty Reads A dirty read happens when a transaction reads data that is being modified by another transaction that has not yet committed.	Transaction A begins. `UPDATE employee SET salary = 31650 WHERE empno = '000090'` Transaction B begins. `SELECT * FROM employee` (Transaction B sees data updated by transaction A. Those updates have not yet been committed.)
Non-Repeatable Reads Non-repeatable reads happen when a query returns data that would be different if the query were repeated within the same transaction. Non-repeatable reads can occur when other transactions are modifying data that a transaction is reading.	Transaction A begins. `SELECT * FROM employee WHERE empno = '000090'` Transaction B begins. `UPDATE employee SET salary = 30100 WHERE empno = '000090'` (Transaction B updates rows viewed by transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.
Phantom Reads Records that appear in a set being read by another transaction. Phantom reads can occur when other transactions insert rows that would satisfy the WHERE clause of another transaction's statement.	Transaction A begins. `SELECT * FROM employee WHERE salary > 30000` Transaction B begins. `INSERT INTO employee (empno, firstnme, midinit, lastname, job, salary) VALUES ('000350', 'NICK', 'A','GREEN','LEGAL COUNSEL',35000)` Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.

The transaction isolation level is a way of specifying whether these transaction anomalies are allowed. The transaction isolation level thus affects the quantity of data locked by a particular transaction. In addition, a DBMS's locking schema might also affect whether these anomalies are allowed. A DBMS can lock either the entire table or only specific rows in order to prevent transaction anomalies.

When Transaction Anomalies Are Possible shows which anomalies are possible under the various locking schemas and isolation levels.

Table 7. When Transaction Anomalies Are Possible

Isolation Level	Table-Level Locking	Row-Level Locking
TRANSACTION_READ_UNCOMMITTED	Dirty reads, nonrepeatable reads, and phantom reads possible	Dirty reads, nonrepeatable reads, and phantom reads possible
TRANSACTION_READ_COMMITTED	Nonrepeatable reads and phantom reads possible	Nonrepeatable reads and phantom reads possible
TRANSACTION_REPEATABLE_READ	Phantom reads not possible because entire table is locked	Phantom reads possible
TRANSACTION_SERIALIZABLE	None	None

The following *java.sql.Connection* isolation levels are supported:

- `TRANSACTION_SERIALIZABLE`

  `RR`, `SERIALIZABLE`, or `REPEATABLE READ` from SQL.

  TRANSACTION_SERIALIZABLE means that Derby treats the transactions as if they occurred serially (one after the other) instead of concurrently. Derby issues locks to prevent all the transaction anomalies listed in Transaction Anomalies from occurring. The type of lock it issues is sometimes called a *range lock*.
- `TRANSACTION_REPEATABLE_READ`

  `RS` from SQL.

  TRANSACTION_REPEATABLE_READ means that Derby issues locks to prevent only dirty reads and non-repeatable reads, but not phantoms. It does not issue range locks for selects.
- `TRANSACTION_READ_COMMITTED`

  `CS` or `CURSOR STABILITY` from SQL.

  TRANSACTION_READ_COMMITTED means that Derby issues locks to prevent only dirty reads, not all the transaction anomalies listed in Transaction Anomalies.

  TRANSACTION_READ_COMMITTED is the default isolation level for transactions.
- `TRANSACTION_READ_UNCOMMITTED`

  `UR`, `DIRTY READ`, or `READ UNCOMMITTED` from SQL.

  For a SELECT INTO, FETCH with a read-only cursor, full select used in an INSERT, full select/subquery in an UPDATE/DELETE, or scalar full select (wherever used), READ UNCOMMITTED allows:

  - Any row that is read during the unit of work to be changed by other application processes.
  - Any row that was changed by another application process to be read even if the change has not been committed by the application process.

  For other operations, the rules that apply to READ COMMITTED also apply to READ UNCOMMITTED.

Configuring isolation levels

If a connection does not specify its isolation level, it inherits the default isolation level for the Derby system. The default value is CS.

When set to CS, the connection inherits the TRANSACTION_READ_COMMITTED isolation level. When set to RR, the connection inherits the TRANSACTION_SERIALIZABLE isolation level, when set to RS, the connection inherits the TRANSACTION_REPEATABLE_READ isolation level, and when set to UR, the connection inherits the TRANSACTION_READ_UNCOMMITTED isolation level.

To override the inherited default, use the methods of *java.sql.Connection*.

In addition, a connection can change the isolation level of the transaction within an SQL statement. For more information, see "SET ISOLATION statement" in the *Java DB Reference Manual*. You can use the WITH clause to change the isolation level for the current statement only, not the transaction. For information about the WITH clause, see "SELECT statement" in the *Java DB Reference Manual*.

In all cases except when you change the isolation level using the WITH clause, changing the isolation level commits the current transaction. In most cases, the current transaction is committed even if you set the isolation level in a way that does not change it (that is, if you set it to its current value). See Isolation levels and concurrency for details.

Note: For information about how to choose a particular isolation level, see "Shielding users from Derby class-loading events" in *Tuning Java DB* and Multi-thread programming tips.

Lock granularity

Derby can be configured for *table-level* locking. With table-level locking, when a transaction locks data in order to prevent any transaction anomalies, it always locks the entire table, not just those rows being accessed.

By default, Derby is configured for row-level locking. Row-level locking uses more memory but allows greater concurrency, which works better in multi-user systems. Table-level locking works best with single-user applications or read-only applications.

You typically set lock granularity for the entire Derby system, not for a particular application. However, at runtime, Derby may escalate the lock granularity for a particular transaction from row-level locking to table-level locking for performance reasons. You have some control over the threshold at which this occurs. For information on turning off row-level locking, see "*derby.storage.rowLocking*" in the *Java DB Reference Manual*. For more information about automatic lock escalation, see "About the system's selection of lock granularity" and "Transaction-based lock escalation" in *Tuning Java DB*. For more information on tuning your Derby system, see "Tuning databases and applications," also in *Tuning Java DB*.

Types and scope of locks in Derby systems

There are several types of locks available in Derby systems, including exclusive, shared, and update locks.

Exclusive locks

When a statement modifies data, its transaction holds an *exclusive* lock on data that prevents other transactions from accessing the data.

This lock remains in place until the transaction holding the lock issues a commit or rollback. Table-level locking lowers concurrency in a multi-user system.

Shared locks

When a statement reads data without making any modifications, its transaction obtains a *shared lock* on the data.

Another transaction that tries to read the same data is permitted to read, but a transaction that tries to update the data will be prevented from doing so until the shared lock is released. How long this shared lock is held depends on the isolation level of the transaction holding the lock. Transactions using the TRANSACTION_READ_COMMITTED isolation level release the lock when the transaction steps through to the next row. Transactions using the TRANSACTION_SERIALIZABLE or TRANSACTION_REPEATABLE_READ isolation level hold the lock until the transaction is committed, so even a SELECT can prevent updates if a commit is never issued. Transactions using the TRANSACTION_READ_UNCOMMITTED isolation level do not request any locks.

Update locks

When a user-defined update cursor (created with the FOR UPDATE clause or by using concurrency mode `ResultSet.CONCUR_UPDATABLE`) reads data, its transaction obtains an *update* lock on the data.

If the user-defined update cursor updates the data, the update lock is converted to an exclusive lock. If the cursor does not update the row, when the transaction steps through to the next row, transactions using the TRANSACTION_READ_COMMITTED isolation level release the lock. (For update locks, the TRANSACTION_READ_UNCOMMITTED isolation level acts the same way as TRANSACTION_READ_COMMITTED.)

Update locks help minimize deadlocks.

Lock compatibility

This table lists compatibility between lock types. + means the lock types are compatible, while - means they are incompatible.

Table 8. Lock Compatibility Matrix

'	Shared	Update	Exclusive
Shared	+	+	-
Update	+	-	-
Exclusive	-	-	-

Scope of locks

The amount of data locked by a statement can vary.

Table locks

A statement can lock the *entire table*.

Table-level locking systems always lock entire tables.

Row-level locking systems can lock entire tables if the WHERE clause of a statement cannot use an index. For example, UPDATES that cannot use an index lock the entire table.

Row-level locking systems can lock entire tables if a high number of single-row locks would be less efficient than a single table-level lock. Choosing table-level locking instead of row-level locking for performance reasons is called *lock escalation*. For more information about this topic, see "About the system's selection of lock granularity" and "Transaction-based lock escalation" in *Tuning Java DB*.

Single-row locks

A statement can lock only *a single row* at a time.

For row-level locking systems:
- For TRANSACTION_REPEATABLE_READ isolation, the locks are released at the end of the transaction.
- For TRANSACTION_READ_COMMITTED isolation, Derby locks rows only as the application steps through the rows in the result. The current row is locked. The row lock is released when the application goes to the next row.
- For TRANSACTION_SERIALIZABLE isolation, however, Derby locks the whole set before the application begins stepping through.
- For TRANSACTION_READ_UNCOMMITTED, no row locks are requested.

Derby locks single rows for INSERT statements, holding each row until the transaction is committed. If there is an index associated with the table, the previous key is also locked.

Range locks

A statement can lock *a range of rows* (range lock).

For row-level locking systems:
- For *any* isolation level, Derby locks *all the rows in the result* plus an entire range of rows for updates or deletes.
- For the TRANSACTION_SERIALIZABLE isolation level, Derby locks all the rows in the result plus an entire range of rows in the table for SELECTs to prevent nonrepeatable reads and phantoms.

For example, if a SELECT statement specifies rows in the *Employee* table where the *salary* is BETWEEN two values, the system can lock more than just the actual rows it returns in the result. It also must lock the entire *range* of rows between those two values to prevent another transaction from inserting, deleting, or updating a row within that range.

An index must be available for a range lock. If one is not available, Derby locks the entire table.

Table 9. Types and scopes of locking

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
Connection.TRANSACTI (SQL: UR)	For SELECT statements, table-level locking is never requested using this isolation level. For other statements, same as for TRANSACTIO	SELECT statements get no locks. For other statements, same as for TRANSACTION_ READ_COMMITTED.
Connection.TRANSACTI (SQL: CS)	SELECT statements get a shared lock on the entire table. The locks are released when the user closes the *ResultSet*. Other statements get exclusive	SELECTs lock and release single rows as the user steps through the *ResultSet*. UPDATEs and DELETEs get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).

Transaction Isolation Level	Table-Level Locking	Row-Level Locking
	locks on the entire table, which are released when the transaction commits.	
Connection.TRANSACTI (SQL: RS)	Same as for TRANS/	SELECT statements get shared locks on the rows that satisfy the WHERE clause (but do not prevent inserts into this range). UPDATEs and DELETEs get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).
Connection.TRANSACTI (SQL: RR)	SELECT statements get a shared lock on the entire table. Other statements get exclusive locks on the entire table, which are released when the transaction commits.	SELECT statements get shared locks on a range of rows. UPDATE and DELETE statements get exclusive locks on a range of rows. INSERT statements get exclusive locks on single rows (and sometimes on the preceding rows).

Notes on locking

In addition to the locks already described, foreign key lookups require briefly held shared locks on the referenced table (row or table, depending on the configuration).

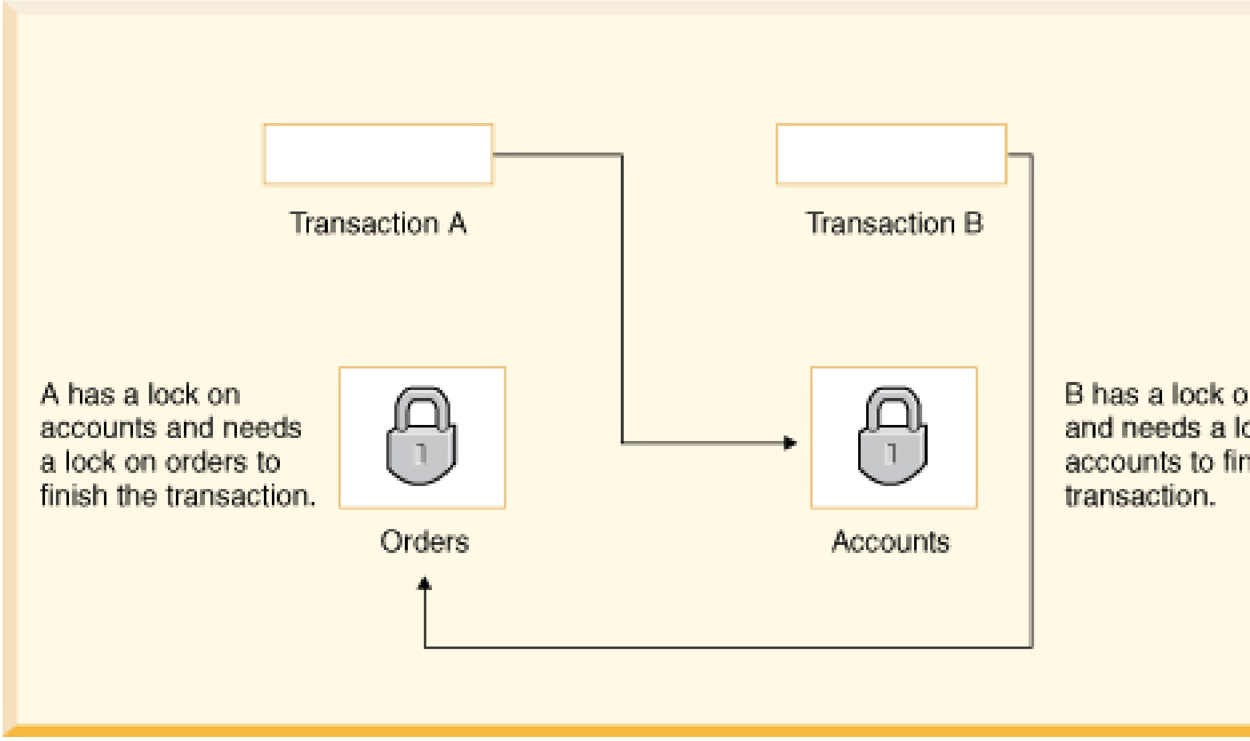
The table and examples in this section do not take performance-based lock escalation into account. Remember that the system can choose table-level locking for performance reasons.

Deadlocks

In a database, a deadlock is a situation in which two or more transactions are waiting for one another to give up locks.

For example, Transaction A might hold a lock on some rows in the *Accounts* table and needs to update some rows in the *Orders* table to finish. Transaction B holds locks on those very rows in the *Orders* table but needs to update the rows in the *Accounts* table held by Transaction A. Transaction A cannot complete its transaction because of the lock on *Orders*. Transaction B cannot complete its transaction because of the lock on *Accounts*. All activity comes to a halt and remains at a standstill forever unless the DBMS detects the deadlock and aborts one of the transactions.

Figure 6. A deadlock where two transactions are waiting for one another to give up locks.



Avoiding deadlocks

Using both row-level locking and the TRANSACTION_READ_COMMITTED isolation level makes it likely that you will avoid deadlocks (both settings are Derby defaults). However, deadlocks are still possible.

Derby application developers can avoid deadlocks by using consistent application logic; for example, transactions that access *Accounts* and *Orders* should always access the tables in the same order. That way, in the scenario described above, Transaction B simply waits for transaction A to release the lock on *Orders* before it begins. When transaction A releases the lock on *Orders*, Transaction B can proceed freely.

Another tool available to you is the LOCK TABLE statement. A transaction can attempt to lock a table in exclusive mode when it starts to prevent other transactions from getting shared locks on a table. For more information, see "LOCK TABLE statement" in the *Java DB Reference Manual*.

Deadlock detection

When a transaction waits more than a specific amount of time to obtain a lock (called the deadlock timeout), Derby can detect whether the transaction is involved in a deadlock.

When Derby analyzes such a situation for deadlocks it tries to determine how many transactions are involved in the deadlock (two or more). Usually aborting one transaction breaks the deadlock. Derby must pick one transaction as the victim and abort that transaction; it picks the transaction that holds the fewest number of locks as the victim, on the assumption that transaction has performed the least amount of work. (This may not be the case, however; the transaction might have recently been escalated from row-level locking to table locking and thus hold a small number of locks even though it has done the most work.)

When Derby aborts the victim transaction, it receives a deadlock error (an *SQLException* with an `SQLState` of 40001). The error message gives you the transaction IDs, the statements, and the status of locks involved in a deadlock situation.

```
ERROR 40001: A lock could not be obtained due to a deadlock,
cycle of locks & waiters is:
Lock : ROW, DEPARTMENT, (1,14)
Waiting XID : {752, X} , APP, update department set location='Boise'
 where deptno='E21'
Granted XID : {758, X} Lock : ROW, EMPLOYEE, (2,8)
Waiting XID : {758, U} , APP, update employee set bonus=150 where
 salary=23840
Granted XID : {752, X} The selected victim is XID : 752
```

For information on configuring when deadlock checking occurs, see Configuring deadlock detection and lock wait timeouts.

Note: Deadlocks are detected only within a single database. Deadlocks across multiple databases are not detected. Non-database deadlocks caused by Java synchronization primitives are not detected by Derby.

Lock wait timeouts

Even if a transaction is not involved in a deadlock, it might have to wait a considerable amount of time to obtain a lock because of a long-running transaction or transactions holding locks on the tables it needs.

In such a situation, you might not want a transaction to wait indefinitely. Instead, you might want the waiting transaction to abort, or *time out*, after a reasonable amount of time, called a *lock wait timeout*.

Configuring deadlock detection and lock wait timeouts

You configure the amount of time a transaction waits before Derby does any deadlock checking with the *derby.locks.deadlockTimeout* property.

You configure the amount of time a transaction waits before timing out with the *derby.locks.waitTimeout* property. When configuring your database or system, you should consider these properties together. For example, in order for any deadlock checking to occur, the *derby.locks.deadlockTimeout* property must be set to a value lower than the *derby.locks.waitTimeout* property. If it is set to a value equal to or higher than the *derby.locks.waitTimeout*, the transaction times out before Derby does any deadlock checking.

By default, *derby.locks.waitTimeout* is set to 60 seconds. -1 is the equivalent of no wait timeout. This means that transactions never time out, although Derby can choose a transaction as a deadlock victim.

Figure 7. One possible configuration: deadlock checking occurs when a transaction has waited 30 seconds; no lock wait timeouts occur.

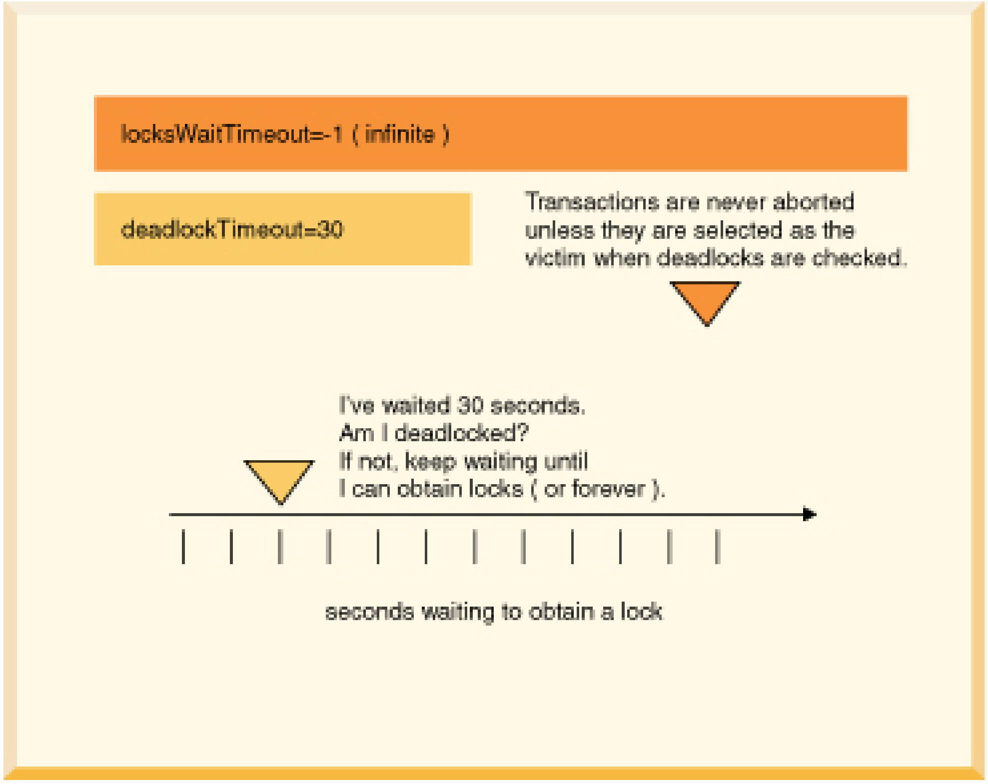


Figure 8. Another typical configuration: deadlock checking occurs after a transaction has waited 60 seconds for a lock; after 90 seconds, the transaction times out and is rolled back.

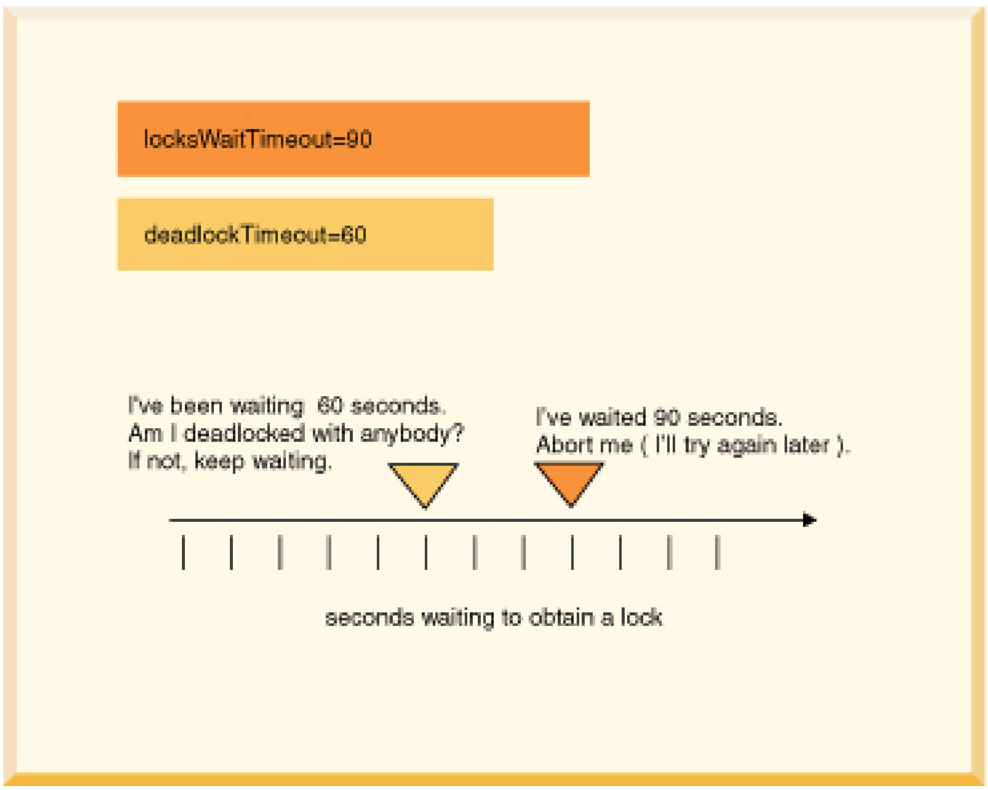
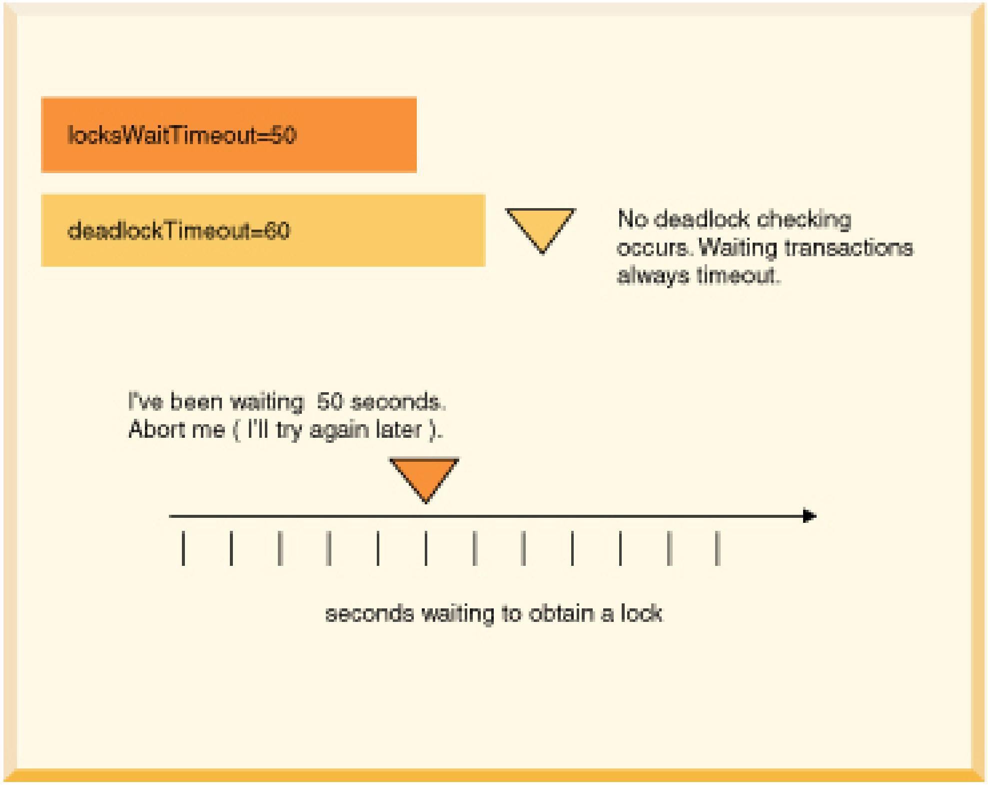


Figure 9. A configuration in which no deadlock checking occurs: transactions time out after they have waited 50 seconds. No deadlock checking occurs.



Debugging Deadlocks

If deadlocks occur frequently in your multi-user system with a particular application, you might need to do some debugging.

Derby provides a class to help you in this situation, *org.apache.derby.diag.LockTable*. You can also set the property *derby.locks.deadlockTrace* to dump additional information to the derby.log file about any deadlocks that occur on your system. See the Tuning Guide for more information on this property. For information, see the *Java DB Server and Administration Guide*.

Programming applications to handle deadlocks

When you configure your system for deadlock and lockwait timeouts and an application could be chosen as a victim when the transaction times out, you should program your application to handle them.

To do this, test for *SQLExceptions* with `SQLStates` of 40001 (deadlock timeout) or 40XL1 or 40XL2 (lockwait timeout).

In the case of a deadlock you might want to re-try the transaction that was chosen as a victim. In the case of a lock wait timeout, you probably do not want to do this right away.

The following code is one example of how to handle a deadlock timeout.

```
/// if this code might encounter a deadlock,
// put the whole thing in a try/catch block
// then try again if the deadlock victim exception
// was thrown
try {
    s6.executeUpdate(
         "UPDATE employee " +
         "SET bonus = 625 "
```

```
        "WHERE empno='000150'");
    s6.executeUpdate("UPDATE project " +
        "SET respemp = '000150' " +
        "WHERE projno='IF1000'");
}
// note: do not catch such exceptions in database-side methods;
// catch such exceptions only at the outermost level of
// application code.
// See Database-side JDBC routines and SQLExceptions.
catch (SQLException se) {
    if (se.getSQLState().equals("40001")) {
        // it was chosen as a victim of a deadlock.
        // try again at least once at this point.
        System.out.println( "Will try the transaction again.");
        s6.executeUpdate("UPDATE employee " +
        "SET bonus = 625 " +
        "WHERE empno='000150'");
        s6.executeUpdate("UPDATE project " +
        "SET respemp = 000150 " +
        "WHERE projno='IF1000'");
    }
    else throw se;
}
```

Working with multiple connections to a single database

This section discusses deploying Derby so that many connections can exist to a single database.

Deployment options and threading and connection modes

A database can be available to multiple connections in several situations.

- Multiple applications access a single database (possible only when Derby is running inside a server framework).
- A single application has more than one *Connection* to the same database.

The way you deploy Derby affects the ways applications can use multi-threading and connections, as shown in Threading and Connection Modes.

Table 10. Threading and Connection Modes

Connection mode	Embedded	Server
Multi-Threaded From an application, using a *single*`Connection` to a Derby database and issuing requests against that connection in multiple threads.	Supply a single *Connection* object to separate threads. Derby ensures that only one operation is applied at a time for consistency. Server frameworks automatically manage multi-threaded operations..	Server frameworks can automatically multi-thread operations. Remote client applications can multi-thread if desired.
Multi-Connection From an application, using multiple connections to a Derby database and issuing requests against those connections on multiple threads.	Create individual connections within a single application and use the appropriate connection for each JDBC request. The connections can all be to the same database, or can be	Remote client applications can establish the multiple connections desired.

Connection mode	Embedded	Server
	to different databases in the same Derby system.	
Multi-User Multiple applications (or JVMs) accessing the same Derby database. Each user application has its own connection or connections to the database.	Not possible. Only one application can access a database at a time, and only one application can access a specific system at a time. When using a pre-1.4 JVM, Derby might not prevent multiple applications from concurrently accessing the same Derby system, but do not allow this because such access can corrupt the databases involved.	Only one server should access a database at a time. Multiple remote client applications can access the same server, and thus can access the same database at the same time through that server.

Multi-user database access

Multi-user database access is possible if Derby is running inside a server framework.

If more than one client application tries to modify the same data, the connection that gets the table first gets the lock on the data (either specific rows or the entire table). The second connection has to wait until the first connection commits or rolls back the transaction in order to access the data. If two connections are only querying and not modifying data, they can both access the same data at the same time because they can each get a shared lock.

Multiple connections from a single application

A single application can work with multiple *Connections* to the same database and assign them to different threads.

You can avoid concurrency and deadlock problems in your application in several ways:

- Use the *TRANSACTION_READ_COMMITTED* isolation level and turn on row-level locking (the defaults).
- Beware of deadlocks caused by using more than one *Connection* in a single thread (the most obvious case). For example, if the thread tries to update the same table from two different `Connections`, a deadlock can occur.
- Assign *Connections* to threads that handle discrete tasks. For example, do not have two threads update the *Hotels* table. Have one thread update the *Hotels* table and a different one update the *Groups* table.
- If threads access the same tables, commit transactions often.
- Multi-threaded Java applications have the ability to self-deadlock without even accessing a database, so beware of that too.
- Use nested connections to share the same lock space.

Working with multiple threads sharing a single connection

JDBC allows you to share a single *Connection* among multiple threads.

Pitfalls of sharing a connection among threads

Here is a review of the potential pitfalls of sharing a single *Connection* among multiple threads.

- Committing or rolling back a transaction closes all open *ResultSet* objects and currently executing *Statements*, unless you are using held cursors.

  If one thread commits, it closes the *Statements* and *ResultSets* of all other threads using the same connection.
- Executing a *Statement* automatically closes any existing open *ResultSet* generated by an earlier execution of that *Statement*.

  If threads share *Statements*, one thread could close another's *ResultSet.*

In many cases, it is easier to assign each thread to a distinct *Connection*. If thread *A* does database work that is not transactionally related to thread *B*, assign them to different *Connections*. For example, if thread *A* is associated with a user input window that allows users to delete hotels and thread *B* is associated with a user window that allows users to view city information, assign those threads to different *Connections*. That way, when thread *A* commits, it does not affect any *ResultSets* or *Statements* of thread B.

Another strategy is to have one thread do queries and another thread do updates. Queries hold shared locks until the transaction commits in SERIALIZABLE isolation mode; use READ_COMMITTED instead.

Yet another strategy is to have only one thread do database access. Have other threads get information from the database access thread.

Multiple threads are permitted to share a *Connection*, *Statement*, or *ResultSet*. However, the application programmer must ensure that one thread does not affect the behavior of the others.

Recommended Practices

Here are some tips for avoiding unexpected behavior:

- Avoid sharing *Statements* (and their *`ResultSets`*) among threads.
- Each time a thread executes a *Statement*, it should process the results before relinquishing the *`Connection`*.
- Each time a thread accesses the *Connection*, it should consistently commit or not, depending on application protocol.
- Have one thread be the "managing" database *Connection* thread that should handle the higher-level tasks, such as establishing the *`Connection`*, committing, rolling back, changing *`Connection`* properties such as auto-commit, closing the *`Connection`*, shutting down the database (in an embedded environment), and so on.
- Close *ResultSets* and *`Statements`* that are no longer needed in order to release resources.

Multi-thread programming tips

You may be sharing a *Connection* among multiple threads because you have experienced poor concurrency using separate transactions.

Here are some tips for increasing concurrency:

- Use row-level locking.
- Use the *TRANSACTION_READ_COMMITTED* isolation level.
- Avoid queries that cannot use indexes; they require locking of all the rows in the table (if only very briefly) and might block an update.

In addition, some programmers might share a statement among multiple threads to avoid the overhead of each thread's having its own. Using the single statement cache, threads can share the same statement from *different connections*. For more information, see "Using the statement cache" in *Tuning Java DB*.

Example of threads sharing a statement

This example shows what can happen if two threads try to share a single *Statement*.

```
PreparedStatement ps = conn.prepareStatement(
    "UPDATE account SET balance =  balance + ? WHERE id = ?");
/* now assume two threads T1,T2 are given this
java.sql.PreparedStatement object and that the following events
happen in the order shown (pseudojava code)*/
T1 - ps.setBigDecimal(1, 100.00);
T1 - ps.setLong(2, 1234);
T2 -  ps.setBigDecimal(1, -500.00);
// *** At this point the prepared statement has the parameters
// -500.00 and 1234
// T1 thinks it is adding 100.00 to account 1234 but actually
// it is subtracting 500.00
T1 - ps.executeUpdate();
T2 - ps.setLong(2, 5678);
// T2 executes the correct update
 T2 - ps.executeUpdate();
/* Also, the auto-commit mode of the connection can lead
to some strange behavior.*/
```

If it is absolutely necessary, the application can get around this problem with Java synchronization.

If the threads each obtain their own *PreparedStatement* (with identical text), their `setXXX` calls do not interfere with each other. Moreover, Derby is able to share the same compiled query plan between the two statements; it needs to maintain only separate state information. However, there is the potential for confusion in regard to the timing of the *commit*, since a single *commit* commits all the statements in a transaction.

Working with database threads in an embedded environment

Do not use `interrupt` calls to notify threads that are accessing a database, because Derby will catch the `interrupt` call and close the connection to the database. Use `wait` and `notify` calls instead.

This will not happen in a client/server environment, but if you want your application to work in either environment it is good practice to follow this rule.

There are also special considerations when working with more than one database thread in an application.

Working with Derby SQLExceptions in an application

JDBC generates exceptions of the type *java.sql.SQLException*. If your application runs on JDK 1.6 or higher, the exceptions will be the refined subtypes of *java.sql.SQLException* introduced by JDBC4. To see the exceptions generated by Derby, retrieve and process the `SQLExceptions` in a catch block.

Information provided in SQL Exceptions

Derby provides the message, *SQLState* values, and error codes in SQL exceptions.

Use the `getSQLState` and `getMessage` methods to view the `SQLState` and error messages. Use `getErrorCode` to see the error code. The error code defines the severity of the error and is not unique to each exception.
Note: Severity is not standardized in Derby. Applications should not depend on the severity returned from SQL exceptions.

Applications should also check for and process *java.sql.SQLWarnings*, which are processed in a similar way. Derby issues an `SQLWarning` if the `create=true` attribute is specified and the database already exists.

Example of processing SQLExceptions

A single error can generate more than one *SQLException*. Use a loop and the `getNextException` method to process all `SQLExceptions` in the chain. In many cases, the second exception in the chain is the pertinent one.

The following is an example:

```
catch (Throwable e) {
    System.out.println("exception thrown:");
    errorPrint(e);
}
static void errorPrint(Throwable e) {
    if (e instanceof SQLException)
        SQLExceptionPrint((SQLException)e);
    else
        System.out.println("A non-SQL error: " + e.toString());
}
static void SQLExceptionPrint(SQLException sqle) {
    while (sqle != null) {
        System.out.println("\n---SQLException Caught---\n");
        System.out.println("SQLState:   " + (sqle).getSQLState());
        System.out.println("Severity: " + (sqle).getErrorCode());
        System.out.println("Message:  " + (sqle).getMessage());
        sqle.printStackTrace();
        sqle = sqle.getNextException();
    }
}
```

If your application runs on JDK 1.4 or higher, then the *SQLException* may wrap another, triggering exception, like an *IOException*. To inspect this additional, wrapped error, call the *SQLException*'s *getCause* method.

See also "Derby Exception Messages and SQL States", in the *Java DB Reference Manual*.

Using Derby as a J2EE resource manager

J2EE, or the Java 2 Platform, Enterprise Edition, is a standard for development of enterprise applications based on reusable components in a multi-tier environment. In addition to the features of the Java 2 Platform, Standard Edition (J2SE), J2EE adds support for Enterprise Java Beans (EJBs), Java Server Pages (JSPs), Servlets, XML and many more. The J2EE architecture is used to bring together existing technologies and enterprise applications in a single, manageable environment.

Derby is a J2EE-conformant component in a distributed J2EE system. As such, it is one part of a larger system that includes, among other things, a JNDI server, a connection pool module, a transaction manager, a resource manager, and user applications. Within this system, Derby can serve as the resource manager.

For more information on J2EE and how to work in this environment, see the J2EE specification available at *http://java.sun.com/j2ee/docs.html*.

Note: This chapter does not show you how to use Derby as a Resource Manager. Instead, it provides details specific to Derby that are not covered in the specification. This information is useful to programmers developing other modules in a distributed J2EE system, not to end-user application developers.

In order to qualify as a resource manager in a J2EE system, J2EE requires these basic areas of support. These three areas of support involve implementation of APIS and are described in "J2EE Compliance: Java Transaction API and javax.sql Extensions" in the *Java DB Reference Manual*.

This chapter describes the Derby classes that implement the APIs and provides some implementation-specific details.

Classes that pertain to resource managers

Derby provides two variants of each *DataSource* interface defined by the JDBC API.

Applications that run on the J2SE 1.4 or 1.5 platform **must** use the first variant. Applications that run on the Java SE 6 platform can use either of the two variants. However, the *DataSource* methods specific to the JDBC 4 API are available only from the second variant (the one whose class name ends with "40").

If an application is running on the Java SE 6 platform, all connection objects returned from the *DataSource* will be JDBC 4 connection objects, regardless of which *DataSource* variant is in use.

The Derby implementation classes for the *DataSource* interfaces are as follows:

- *org.apache.derby.jdbc.EmbeddedDataSource* and *org.apache.derby.jdbc.EmbeddedDataSource40*

  Implements the *javax.sql.DataSource* interface, which a JNDI server can reference. Typically this is the object that you work with as a *DataSource*.
- *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource* and *org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40*

  Implements the *javax.sql.ConnectionPoolDataSource* interface. A factory for *PooledConnection* objects.
- *org.apache.derby.jdbc.EmbeddedXADataSource* and *org.apache.derby.jdbc.EmbeddedXADataSource40*

  Derby's implementation of the *javax.sql.XADataSource* interface.

See the javadoc for each class for more information.

Getting a DataSource

Normally, you can simply work with the interfaces for *javax.sql.DataSource*, *javax.sql.ConnectionPoolDataSource*, and *javax.sql.XADataSource*, as shown in the following examples.

```
// If your application is running on the Java SE 6 platform,
// and if you would like to call DataSource methods specific
// to the JDBC 4 API (for example, isWrapperFor), use the
// JDBC 4 variants of these classes:
//
//   org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource40
//   org.apache.derby.jdbc.EmbeddedDataSource40
//   org.apache.derby.jdbc.EmbeddedXADataSource40
//
import org.apache.derby.jdbc.EmbeddedConnectionPoolDataSource;
import org.apache.derby.jdbc.EmbeddedDataSource;
import org.apache.derby.jdbc.EmbeddedXADataSource;

javax.sql.ConnectionPoolDataSource cpds = new
 EmbeddedConnectionPoolDataSource();
javax.sql.DataSource ds = new EmbeddedDataSource();
javax.sql.XADataSource xads = new EmbeddedXADataSource();
```

Derby provides six properties for a *DataSource*. These properties are in *org.apache.derby.jdbc.EmbeddedDataSource*. They are:

- *DatabaseName*

  This mandatory property must be set. It identifies which database to access. To access a database named *wombat* located at */local1/db/wombat*, call *setDatabaseName("/local1/db/wombat")* on the data source object.
- *CreateDatabase*

  Optional. Sets a property to create a database the next time the *getConnection* method of a data source object is called. The string *createString* is always "create" (or possibly null). (Use the method *setDatabaseName()* to define the name of the database.)
- *ShutdownDatabase*

  Optional. Sets a property to shut down a database. The string *shutDownString* is always "shutdown" (or possibly null). Shuts down the database the next time the *getConnection* method of a data source object is called.
- *DataSourceName*

  Optional. Name for *ConnectionPoolDataSource* or *XADataSource*. Not used by the data source object. Used for informational purposes only.
- *Description*

  Optional. Description of the data source. Not used by the data source object. Used for informational purposes only.
- *connectionAttributes*

  Optional. Connection attributes specific to Derby. See the *Java DB Reference Manual* for a more information about the attributes.

Shutting down or creating a database

If you need to shut down or create a database, it is easiest just to work with the Derby-specific implementations of interfaces, as shown in these examples.

```
javax.sql.XADataSource xads = makeXADataSource(mydb, true);


// example of setting property directory using
// Derby 's XADataSource object
import org.apache.derby.jdbc.EmbeddedXADataSource;
import javax.sql.XADataSource;
// dbname is the database name
// if create is true, create the database if not already created
XADataSource makeXADataSource (String dbname, boolean create)
{
    //
    // If your application runs on JDK 1.6 or higher, then
    // you will use the JDBC4 variant of this class:
    // EmbeddedXADataSource40.
    //
    EmbeddedXADataSource xads = new EmbeddedXADataSource();
    // use Derby 's setDatabaseName call
    xads.setDatabaseName(dbname);
    if (create)
        xads.setCreateDatabase("create");
    return xads;
}
```

Setting the property does not create or shut down the database. The database is not actually created or shut down until the next connection request.

Derby and Security

Derby can be deployed in a number of ways and in a number of different environments. The security needs of the Derby system are also diverse.

Derby supplies or supports the following optional security mechanisms:

- *User authentication*

  Derby verifies user names and passwords before permitting them access to the Derby system.
- *User authorization*

  A means of granting specific users permission to read a database or to write to a database.
- *Disk encryption*

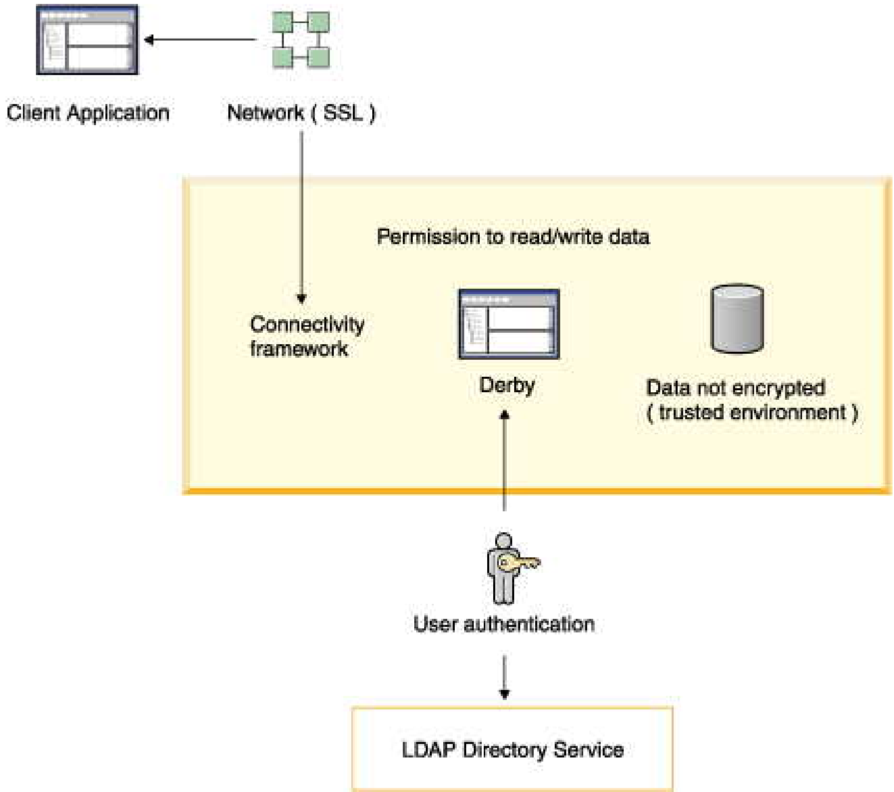
  A means of encrypting Derby data stored on disk.
- *Validation of Certificate for Signed Jar Files*

  In a Java 2 environment, Derby validates certificates for classes loaded from signed jar files.
- *Network encryption and authentication*

  Derby network traffic may be encrypted with SSL/TLS. SSL/TLS certificate authentication is also supported. See *"Network encryption and authentication with SSL/TLS"* in the *Java DB Server and Administration Guide* for details.

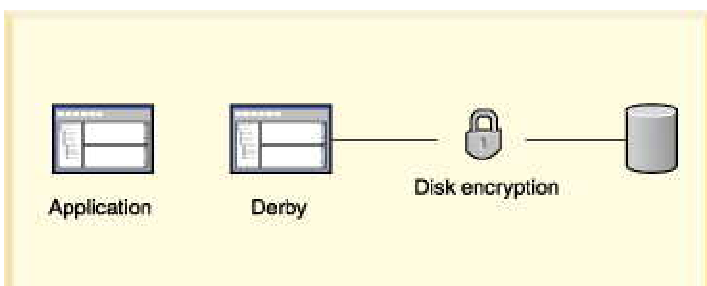
The following figure shows some of the Derby security mechanisms at work in a client/server environment. User authentication is performed by accessing an LDAP Directory Service. The data in the database is not encrypted in this trusted environment.

Figure 10. Example of using an LDAP Directory Service in a trusted environment.



The following figure shows how another Derby security mechanism, disk encryption, protects data when the recipient might not know how to protect data. It is useful for databases deployed in an embedded environment.

Figure 11. Example of using disk encryption to protect data.



Configuring security for your environment

In most cases, you enable Derby's security features through the use of properties. It is important to understand the best way of setting properties for your environment.

Derby does not come with a built-in superuser. For that reason, be careful when configuring Derby for user authentication and user authorization.

1. When first working with security, work with system-level properties only so that you can easily override them if you make a mistake.
2. Be sure to create at least one valid user, and grant that user full (read-write) access. For example, you might always want to create a user called *sa* with the password *derby* while you are developing.

3. Test the authentication system while it is still configured at the system level. Be absolutely certain that you have configured the system correctly before setting the properties as database-level properties.
4. Before disabling system-level properties (by setting *derby.database.propertiesOnly* to true), test that at least one database-level read-write user (such as *sa*) is valid. If you do not have at least one valid user that the system can authenticate, you will not be able to access your database.

Configuring security in a client/server environment

This procedure requires a system with multiple databases and some administrative resources.

1. Configure security features as system properties. See Scope of properties and Setting system-wide properties.
2. Provide administrative-level protection for the *derby.properties* file and Derby databases. For example, you can protect these files and directories with operating system permissions and firewalls.
3. Turn on user authentication for your system. All users must provide valid user IDs and passwords to access the Derby system. If you are using Derby's built-in users, configure users for the system in the *derby.properties* file. Provide the protection for this file.
4. Configure user authorization for sensitive databases in your system. Only designated users will be able to access sensitive databases. You typically configure user authorization with database-level properties. It is also possible to configure user authorization with system-level properties. This is useful when you are developing systems or when all databases have the same level of sensitivity.
5. Check and if necessary configure your Derby network security according to your environment. See the section "Network client security" in the *Java DB Server and Administration Guide*.

Configuring security in an embedded environment

In an embedded environment, typically there is only one database per system and there are no administrative resources to protect databases.

To configure security in an embedded environment:

1. Encrypt the database when you create it.
2. Configure all security features as database-level properties. These properties are stored in the database (which is encrypted). See Scope of properties and Setting database-wide properties for more information.
3. Turn on protection for database-level properties so that they cannot be overridden by system properties by setting the *derby.database.propertiesOnly* property to TRUE. See the *Java DB Reference Manual* for details on this property.
4. To prevent unauthorized users from accessing databases once they are booted, turn on user authentication for the database and configure user authorization for the database.
5. If you are using Derby's built-in users, configure each user as a database-level property so that user names and passwords can be encrypted.

Working with user authentication

Derby provides support for user authentication. User authentication means that Derby authenticates the name and password for a user before allowing that user access to the system.

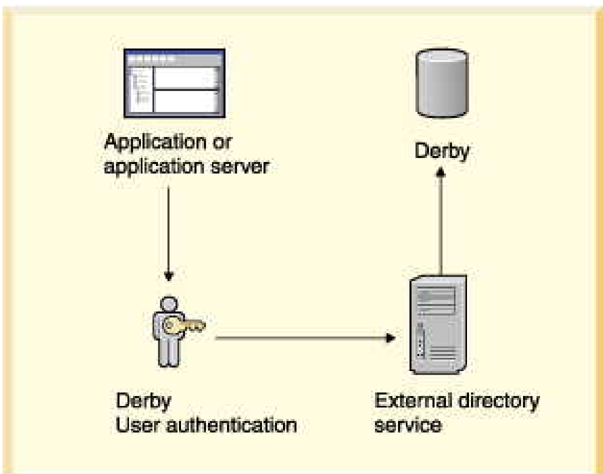
When user authentication is enabled (which it is not by default), the user requesting a connection must provide a valid name and password, which Derby verifies against the repository of users defined for the system. After Derby authenticates the user, it grants the user access to the Derby system but not necessarily access to the database made in the connection request. In the Derby system, access to a database is determined by user authorization.

For user authentication, Derby allows you to provide a repository of users in a number of different ways. For example, you can hook Derby up to an external directory service elsewhere in your enterprise, create your own directory service, or use Derby's simple mechanism for creating a built-in repository of users.

You can define a repository of users for a particular database or for an entire system, depending on whether you use system-wide or database-wide properties.

When Derby user authentication is enabled and Derby uses an external directory service, the architecture looks something like that shown in the figure below.

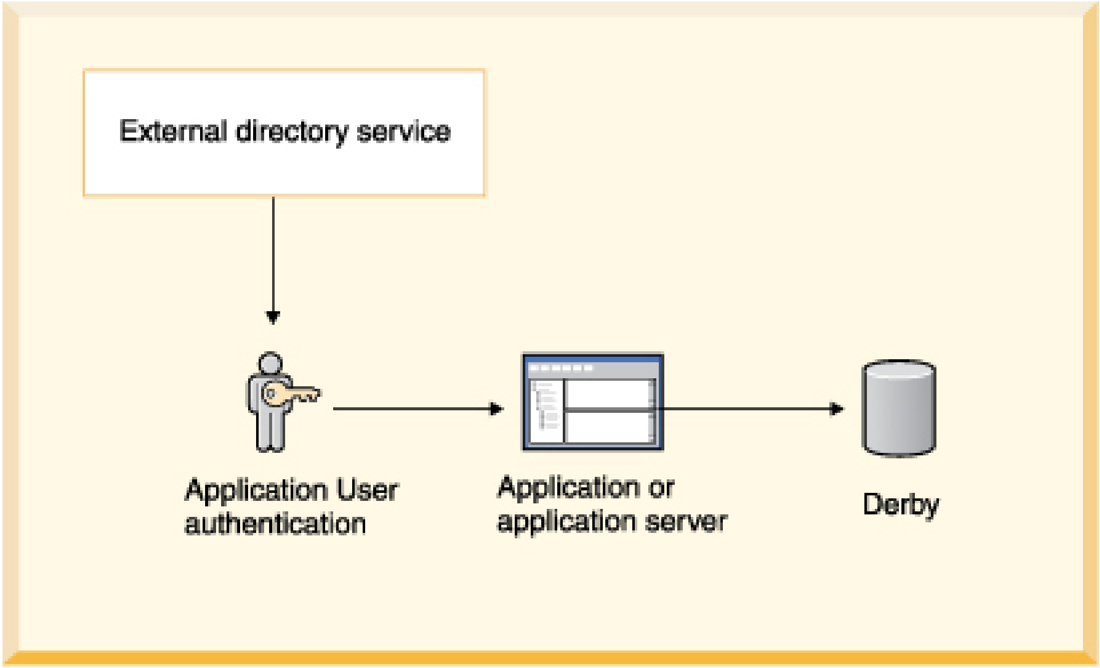
Figure 12. Derby user authentication using an external service. The application can be a single-user application with an embedded Derby engine or a multi-user application server.



Derby always runs embedded in another Java application, whether that application is a single-user application or a multiple-user application server or connectivity framework.

A database can be accessed by only one JVM at a time, so it is possible to deploy a system in which the application in which Derby is embedded, not Derby, handles the user authentication by connecting to an external directory service.

Figure 13. The application provides the user authentication using an external service. The application can be a single-user application with an embedded Derby engine or a multi-user application server.



Enabling user authentication

To enable user authentication, set the *derby.connection.requireAuthentication* property to true. Otherwise, Derby does not require a user name and password. You can set this property as a system-wide property or as a database-wide property.

For a multi-user product, you would typically set it for the system in the *derby.properties* file for your server, since it is in a trusted environment.

Note: If you start a Derby system with user authentication enabled but without defining at least one user, you will not be able to shut down the system gracefully. When Derby is running in a connectivity server and user authentication is turned on, stopping the server requires a user name and password. You will need to alter shutdown scripts accordingly.

Note: Additionally, if you create and start a Derby system with user authentication and SQL authorization both enabled, or plan to enable them later, you should make sure you create the database by connecting as the user that is to become the database owner. If you neglect to supply a user when the database is created, the database owner will by default become "APP". If you later enable both authentication and SQL authorization and "APP" is a not valid user name, you will not be able to perform operations restricted to the database owner, including shutting down the database (as opposed to the full system which may currently be shut down by any authenticated user, see previous note). Nor will you be able to (re)encrypt the database nor perform a full upgrade of it.

Defining users

Derby provides several ways to define the repository of users and passwords. To specify which of these services to use with your Derby system, set the property *derby.authentication.provider* to the appropriate value as discussed here.

Setting the property as a system-wide property creates system-wide users. Setting the property as a database-wide property creates users for a single database only.

- External directory service: LDAP directory service. This includes Windows NT domain user authentication through the Netscape NT Synchronization Service.

- User-defined class
- Built-in Derby users

Note: Shutting down the Derby system (for example, using the *shutdown=true* form of the connection URL without specifying a particular database) when user authentication is turned on requires that you define at least one user as a system-wide user.

External directory service

A directory service stores names and attributes of those names. A typical use for a directory service is to store user names and passwords for a computer system. Derby uses the Java naming and directory interface (JNDI) to interact with external directory services that can provide authentication of users' names and passwords.

LDAP directory service

You can allow Derby to authenticate users against an existing LDAP directory service within your enterprise. LDAP (lightweight directory access protocol) provides an open directory access protocol running over TCP/IP. An LDAP directory service can quickly authenticate a user's name and password.

To use an LDAP directory service, set *derby.authentication.provider* to *LDAP*.

Examples of LDAP service providers are:

- Netscape Directory Server

  Netscape Directory Server is an LDAP directory server. In addition, the Netscape Directory Synchronization Service synchronizes entries in a Windows NT directory with the entries in Netscape's Directory Server. It allows you to use the Windows NT directory as a repository for Derby users.
- UMich slapd (freeware for the UNIX platform from the University of Michigan)
- AE SLAPD for Windows NT, from AEInc

Libraries for LDAP user authentication:

To use an LDAP directory service with Derby, you need these libraries in your classpath.

- *jndi.jar*

  JNDI classes
- *ldap.jar*

  LDAP provider from Sun
- *providerutil.jar*

  JNDI classes for a provider

Derby does not provide these libraries; they are available from Sun on the JNDI page. Use the 1.1.x versions of these libraries, not the 1.2.x versions. You might need to do two separate downloads to obtain all the required libraries.

Setting up Derby to use your LDAP directory service:

When specifying LDAP as your authentication service, you must specify what LDAP server to use.

- *derby.authentication.server*

  Set the property *derby.authentication.server* to the URL to the LDAP server. For example:

  ```
  derby.authentication.server=ldap://godfrey:389/
  ```

  The LDAP server may be specified using just the server name, the server name and its port number separated by a colon, or a ldap URL. If a full URL is not provided,

Derby will by default use unencrypted LDAP - to use SSL encrypted LDAP an URL starting with "ldaps://" must be provided.

Also note that support for ldaps:// URLs requires that Derby runs on Java 1.4.2 or higher.

Guest access to search for DNs:

In an LDAP system, users are hierarchically organized in the directory as a set of entries. An *entry* is a set of name-attribute pairs identified by a unique name, called a DN (distinguished name).

An entry is unambiguously identified by a DN, which is the concatenation of selected attributes from each entry in the tree along a path leading from the root down to the named entry, ordered from right to left. For example, a DN for a user might look like this:

```
cn=mary,ou=People,o=FlyTours.com

uid=mary,ou=People,o=FlyTours.com
```

The allowable entries for the name are defined by the entry's *objectClass*.

An LDAP client can bind to the directory (successfully log in) if it provides a user ID and password. The user ID must be a DN, the fully qualified list of names and attributes. This means that the user must provide a very long name.

Typically, the user knows only a simple user name (e.g., the first part of the DN above, *mary*). With Derby, you do not need the full DN, because an LDAP client (Derby) can go to the directory first as a guest or even an anonymous user, search for the full DN, then rebind to the directory using the full DN (and thus authenticate the user).

Derby typically initiates a search for a full DN before binding to the directory using the full DN for user authentication. Derby does not initiate a search in the following cases:

- You have set *derby.authentication.ldap.searchFilter* to *derby.user*.
- A user DN has been cached locally for the specific user with the *derby.user.UserName* property.

For more information, see "*derby.authentication.ldap.searchFilter*" in the *Java DB Reference Manual*.

Some systems permit anonymous searches; other require a user DN and password. You can specify a user's DN and password for the search with the properties listed below. In addition, you can limit the scope of the search by specifying a filter (definition of the object class for the user) and a base (directory from which to begin the search) with the properties listed below.

- *derby.authentication.ldap.searchAuthDN (optional)*

  Specifies the DN with which to bind (authenticate) to the server when searching for user DNs. This parameter is optional if anonymous access is supported by your server. If specified, this value must be a DN recognized by the directory service, and it must also have the authority to search for the entries.

  If not set, it defaults to an anonymous search using the root DN specified by the *derby.authentication.ldap.searchBase* property. For example:

  ```
  uid=guest,o=FlyTours.com
  ```
- *derby.authentication.ldap.searchAuthPW (optional)*

  Specifies the password to use for the guest user configured above to bind to the directory service when looking up the DN. If not set, it defaults to an anonymous search using the root DN specified by the *derby.authentication.ldap.searchBase* property.

```
myPassword
```

- *derby.authentication.ldap.searchBase (optional)*

  Specifies the root DN of the point in your hierarchy from which to begin a guest search for the user's DN. For example:

  ```
  ou=people,o=FlyTours.com
  ```

  When using Netscape Directory Server, set this property to the root DN, the special entry to which access control does not apply (optional).

To narrow the search, you can specify a user's *objectClass*.

- *derby.authentication.ldap.searchFilter (optional)*

  Set *derby.authentication.ldap.searchFilter* to a logical expression that specifies what constitutes a user for your LDAP directory service. The default value of this property is `objectClass=inetOrgPerson`. For example:

  ```
  objectClass=person
  ```

See the *Java DB Reference Manual* for details on all these properties.

LDAP performance issues:

For performance reasons, the LDAP directory server should be in the same LAN as Derby. Derby does not cache the user's credential information locally and thus must connect to the directory server every time a user connects.

Connection requests that provide the full DN are faster than those that must search for the full DN.

Considerations when using Windows NT with LDAP:

Netscape provides LDAP functionality for Windows NT systems with its Netscape Directory Synchronization service, which synchronizes the Windows NT users with the Netscape Directory Server. SSL is recommended in this configuration.

LDAP restrictions:

Derby does not support LDAP groups.

JNDI-specific properties for external directory services

Derby allows you to set a few advanced JNDI properties, which you can set in any of the supported ways of setting Derby properties. Typically you would set these at the same level (database or system) for which you configured the external authentication service.

The list of supported properties can be found in Appendix A: JNDI Context Environment in the Java Naming and Directory API at *http://java.sun.com/products/jndi/reference/api/index.html*. The external directory service must support the property.

Each JNDI provider has its set of properties that you can set within the Derby system.

For example, you can set the property *java.naming.security.authentication* to allow user credentials to be encrypted on the network if the provider supports it. You can also specify that SSL be used with LDAP (LDAPS).

User-defined class

Set *derby.authentication.provider* to the full name of a class that implements the public interface *org.apache.derby.authentication.UserAuthenticator*.

By writing your own class that fulfills some minimal requirements, you can hook Derby up to an *external authentication service* other than LDAP. To do so, specify an external

authentication service by setting the property `derby.authentication.provider` to a class name that you want Derby to load at startup.

The class that provides the external authentication service must implement the public interface *org.apache.derby.authentication.UserAuthenticator* and throw exceptions of the type *java.sql.SQLException* where appropriate.

Using a user-defined class makes Derby adaptable to various naming and directory services.

Example of setting a user-defined class:

A very simple example of a class that implements the *org.apache.derby.authentication.UserAuthenticator* interface.

```
import org.apache.derby.authentication.UserAuthenticator;
import java.io.FileInputStream;
import java.util.Properties;
import java.sql.SQLException;
/**
  * A simple example of a specialized Authentication scheme.
  * The system property 'derby.connection.requireAuthentication'
  * must be set
  * to true and 'derby.authentication.provider' must
  * contain the full class name of the overriden authentication
  * scheme,  i.e., the name of this class.
  *
  * @see org.apache.derby.authentication.UserAuthenticator
  */

public class MyAuthenticationSchemeImpl implements
UserAuthenticator {
    private static final String USERS_CONFIG_FILE = "myUsers.cfg";
    private static Properties usersConfig;

    // Constructor
    // We get passed some Users properties if the
    //authentication service could not set them as
    //part of System properties.
    //
    public MyAuthenticationSchemeImpl() {
    }
    /* static block where we load the users definition from a
users configuration file.*/

    static {
       /* load users config file as Java properties
        File must be in the same directory where
        Derby  gets started.
       (otherwise full path must be specified) */
       FileInputStream in = null;
       usersConfig = new Properties();
       try {
           in = new FileInputStream(USERS_CONFIG_FILE);
           usersConfig.load(in);
           in.close();
       } catch (java.io.IOException ie) {
           // No Config file. Raise error message
           System.err.println(
             "WARNING: Error during Users Config file
retrieval");
           System.err.println("Exception: " + ie);
       }
    }
    /**
     * Authenticate the passed-in user's credentials.
     * A more complex class could make calls
     * to any external users directory.
     *
```

```
     * @param userName               The user's name
     * @param userPassword           The user's password
     * @param databaseName           The database
     * @param infoAdditional jdbc connection info.
     * @exception SQLException on failure
     */
    public boolean authenticateUser(String userName,
     String userPassword,
     String databaseName,
     Properties info)
       throws SQLException
    {
       /* Specific Authentication scheme logic.
        If user has been authenticated, then simply return.
        If user name and/or password are invalid,
        then raise the appropriate exception.

       This example allows only users defined in the
       users config properties object.

       Check if the passed-in user has been defined for the system.
       We expect to find and match the property corresponding to
       the credentials passed in. */
       if (userName == null)
           // We do not tolerate 'guest' user for now.
           return false;
       //
       // Check if user exists in our users config (file)
       // properties set.
       // If we did not find the user in the users config set, then
       // try to find if the user is defined as a System property.
       //
       String actualUserPassword;
       actualUserPassword = usersConfig.getProperty(userName);
       if (actualUserPassword == null)
           actualUserPassword = System.getProperty(userName);
       if (actualUserPassword == null)
            // no such passed-in user found
            return false;
            // check if the password matches
       if (!actualUserPassword.equals(userPassword))
       return false;
       // Now, check if the user is a valid user of the database
       if (databaseName != null)
       {
             /* if database users restriction lists present, then check
            if there is one for this database and if so,
            check if the user is a valid one of that database.
            For this example, the only user we authorize in database
            DarkSide is user 'DarthVader'. This is the only database
            users restriction list we have for this example.
            We authorize any valid (login) user to access the
            OTHER databases in the system.
            Note that database users ACLs could be set in the same
            properties file or a separate one and implemented as you
            wish. */
            //
           if (databaseName.equals("DarkSide")) {
              // check if user is a valid one.
              if (!userName.equals("DarthVader"))
                  // This user is not a valid one of the passed-in
                  return false;
           }
       }
       // The user is a valid one in this database
       return true;
    }
}
```

Built-in Derby users

Derby provides a simple, built-in repository of user names and passwords.

To use the built-in repository, set *derby.authentication.provider* to *BUILTIN*. Using built-in users is an alternative to using an external directory service such as LDAP.

```
derby.authentication.provider=BUILTIN
```

You can create user names and passwords for Derby users by specifying them with the *derby.user.UserName* property.

Note: These user names are case-sensitive for user authentication. User names are *SQL92Identifiers*. Delimited identifiers are allowed:

```
derby.user."FRed"=java
```

Note: For passwords, it is a good idea not to use words that would be easily guessed, such as a login name or simple words or numbers. A password should be a mix of numbers and upper- and lowercase letters.

Database-level properties

When you create users with database-level properties, those users are available to the specified database only.

You set the property once for each user. To delete a user, set that user's password to null.

```
-- adding the user sa with password 'derbypass'
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.sa', 'derbypass')
-- adding the user mary with password 'little7xylamb'
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.mary', 'little7xylamb')
-- removing mary by setting password to null
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.mary', null)
```

System-level properties

When you create users with system-level properties, those users are available to all databases in the system.

You set the value of this system-wide property once for each user, so you can set it several times. To delete a user, remove that user from the file.

You can define this property in the usual ways -- typically in the *derby.properties* file. For more information about setting properties, see Working with Derby properties.

Here is a sample excerpt from the *derby.properties* file:

```
# Users definition
#
derby.user.sa=derbypass
derby.user.mary=little7xylamb
```

List of user authentication properties

This table summarizes the various properties related to user authentication.

Table 11. User authentication properties

Property Name	Use
derby.connection.requireAuthentication	Turns on user authentication.

Property Name	Use
derby.authentication.provider	Specifies the kind of user authentication to use.
derby.authentication.server	For LDAP user authentication, specifies the location of the server.
derby.authentication.ldap.searchAuthDN, *derby.authentication.ldap.searchAuthPW*, *derby.authentication.ldap.searchFilter*, and *derby.authentication.ldap.searchBase*	Configures the way that DN searches are performed.
derby.user.UserName	Creates a user name and password for the built-in user repository inDerby.
*java.naming.**	JNDI properties. See Appendix A in the JNDI API reference for more information about these properties.

Programming applications for Derby user authentication

This section discusses programming user authentication into applications for use with Derby.

Programming the application to provide the user and password

In the *DriverManager.getConnection* call, an application can provide the user name and password in the following ways.

- Separately as arguments to the following signature of the method: *getConnection(String url, String user, String password)*

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", "mary", "little7xylamb");
```

- As attributes to the database connection URL

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB;user=mary;password=little7xylamb");
```

- By setting the user and password properties in a Properties object as with other connection URL attributes

```
Properties p = new Properties();
p.put("user", "mary");
p.put("password", "little7xylamb");
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", p);
```

Note: The password is not encrypted. When you are using Derby in the context of a server framework, the framework should be responsible for encrypting the password across the network. If your framework does not encrypt the password, consider using SSL.

For information about the treatment of user names within the Derby system, see Users and authorization identifiers.

Login failure exceptions with user authentication

If user authentication is turned on and a valid user name and password are not provided, *SQLException* 08004 is raised.

```
ERROR 08004: Connection refused : Invalid authentication.
```

Users and authorization identifiers

User names within the Derby system are known as *authorization identifiers*. The authorization identifier is a string that represents the name of the user, if one was provided in the connection request.

For example, the built-in function CURRENT_USER returns the authorization identifier for the current user.

Once the authorization identifier is passed to the Derby system, it becomes an *SQL92Identifier*. `SQL92Identifiers`-the kind of identifiers that represent database objects such as tables and columns-are case-insensitive (they are converted to all caps) unless delimited with double quotes, are limited to 128 characters, and have other limitations.

User names must be valid authorization identifiers even if user authentication is turned off, and even if all users are allowed access to all databases.

For more information about *SQL92Identifiers*, see the *Java DB Reference Manual*.

Authorization identifiers, user authentication, and user authorization

When working with both user authentication and user authorization, you need to understand how user names are treated by each system.

If an external authentication system is used, the conversion of the user's name to an authorization identifier does not happen until *after* authentication has occurred but *before* user authorization. Imagine, for example, a user named Fred.

- Within the user authentication system, Fred is known as *FRed*. Your external user authorization service is case-sensitive, so Fred must always type his name that way.

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", "FRed", "flintstone");
```
- Within the Derby user authorization system, Fred becomes a case-insensitive authorization identifier. Fred is known as *FRED*.
- When specifying which users are authorized to access the accounting database, you must list Fred's authorization identifier, *FRED* (which you can type as *FRED*, *FREd*, or *fred*, since the system automatically converts it to all-uppercase).

```
derby.fullAccessUsers=sa,FRED,mary
```

Let's take a second example, where Fred has a slightly different name within the user authentication system.

- Within the user authentication system, Fred is known as *Fred!*. You must now put double quotes around the name, because it is not a valid `SQL92Identifier`. (Derby knows to remove the double quotes when passing the name to the external authentication system.)

```
Connection conn = DriverManager.getConnection(
    "jdbc:derby:myDB", "\"Fred!\"", "flintstone");
```
- Within the Derby user authorization system, *Fred* becomes a case-sensitive authorization identifier. Fred is known as *Fred!*.
- When specifying which users are authorized to access the accounting database, you must list Fred's authorization identifier, *"Fred!"* (which you must always delimit with double quotation marks).

```
derby.fullAccessUsers=sa,"Fred!",manager
```

As shown in the first example, your external authentication system may be case-sensitive, whereas the authorization identifier within Derby may not be. If your authentication system allows two distinct users whose names differ by case, delimit all user names within the connection request to make all user names case-sensitive within the Derby system. In addition, you must also delimit user names that do not conform to *SQL92Identifier* rules with double quotes.

Database owner

The term *database owner* refers to the current authorization identifier when the database is created, that is, the user creating the database. If you enable or plan to enable SQL authorization, controlling the identity of the database owner becomes important.

When a database is created, the database owner of that database gets implicitly set to the authorization identifier used in the connect operation which creates the database, for example by supplying the URL attribute "user". Note that this applies even if authentication is not (yet) enabled. In SQL, the built-in functions USER and the equivalent CURRENT_USER return the current authorization identifier.

If the database is created *without* supplying a user (only possible if authentication is not enabled), the database owner gets set to the default authorization identifier, "APP", which is also the name of the default schema, see the section "SET SCHEMA statement" in the *Java DB Reference Manual*.

The database owner has automatic SQL level permissions when SQL authorization is enabled, see more about this in User authorizations.

To further enhance security, when *both*authentication and SQL authorization are enabled for a database, Derby restricts some special powers to the database owner: only the database owner is allowed to shut down the database, to encrypt or reencrypt the database or to perform a full upgrade of it. These powers can not be delegated.

Attention: There is currently no way of changing the database owner once the database is created. This means that if you plan to run with SQL authorization enabled, you should make sure to create the database as the user you want to be the owner.

User names and schemas

User names can affect a user's default schema.

For information about user names and schemas, see "SET SCHEMA statement" in the *Java DB Reference Manual.*

Exceptions when using authorization identifiers

Specifying an invalid authorization identifier in a database user authorization property raises an exception. Specifying an invalid authorization identifier in a connection request raises an exception.

User authorizations

When you specify user authorizations, Derby verifies that a user has been granted permission to access a system, database, object, or SQL action.

There are two types of user authorization in Derby, connection authorization and SQL authorization. Connection authorization specifies the access that users have to connect to a system or database. SQL authorization controls the permissions that users have on

database objects or for SQL actions. You can set the user authorization properties in Derby as system-level properties or database-level properties.

Set system-level user authorizations when you are developing applications, or when you want to specify a secure default authorization for all users to connect to all of the databases in the system.

There are several properties that you can set to control database-level user authorizations. Some of the properties are general properties that set the access mode for all users. Other properties are user specific properties that set the type of access for specific user IDs.

The properties that affect authorization are:

- The `derby.database.defaultConnectionMode` property controls the default access mode. Use the `derby.database.defaultConnectionMode` property to specify the default connection access that users have when they connect to the database. If you do not explicitly set the `derby.database.defaultConnectionMode` property, the default user authorization for a database is *fullAccess*, which is read-write access.
- The `derby.database.fullAccessUsers` and `derby.database.readOnlyAccessUsers` properties are user specific properties. Use these properties to specify the user IDs that have read-write access and read-only access to a database.
- The `derby.database.sqlAuthorization` property enables SQL standard authorization. Use the `derby.database.sqlAuthorization` property to specify if object owners can grant and revoke permission for users to perform SQL actions on database objects. The default setting for the `derby.database.sqlAuthorization` property is *FALSE*. When the `derby.database.sqlAuthorization` property is set to *TRUE*, object owners can use the GRANT and REVOKE SQL statements to set the user permissions for specific database objects or for specific SQL actions.

If you do not specify the user authorizations for a specific user ID, the user ID inherits whatever authorization is set as the default user authorization for the database.

Tip: If you set the `derby.database.defaultConnectionMode` property to *noAccess* or *readOnlyAccess*, you should allow at least one user read-write access. Otherwise, depending on the default connection authorization you specify, you will configure the database so that it cannot be accessed or changed.

How user authorization properties work together

The `derby.database.defaultConnectionMode` property and the `derby.database.sqlAuthorization` property work together. The default settings for these properties allow anyone to access and drop the database objects that you create. You can change the default access mode by specifying different settings for these properties.

- When the `derby.database.sqlAuthorization` property is *FALSE*, the ability to read from or write to database objects is determined by the setting for the `derby.database.defaultConnectionMode` property. If the `derby.database.defaultConnectionMode` property is set to *readOnlyAccess*, users can access all of the database objects but they cannot update or drop the objects.
- When the `derby.database.sqlAuthorization` property is *TRUE*, the ability to read from or write to database objects is further restricted to the owner of the database objects. The owner must grant permission for others to access the database objects. No one but the owner of an object or the database owner can drop the object.
- The access mode specified for the `derby.database.defaultConnectionMode` property overrides the permissions that are granted by the owner of a database

object. For example, if a user is granted INSERT privileges on a table but the user only has read-only connection authorization, the user cannot insert data into the table.

Changes to connection authorization settings

Connection authorization properties are fixed for the duration of a connection. If you change the connection authorization properties during a connection, those changes are not in affect until you establish a new connection.

Setting the default connection access mode

Use the `derby.database.defaultConnectionMode` property to specify the default type of access that users have when they connect to the database.

The valid settings for the `derby.database.defaultConnectionMode` property are:

- *noAccess*
- *readOnlyAccess*
- *fullAccess*

If you do not specify a setting for the `derby.database.defaultConnectionMode` property, the default access setting is *fullAccess*.

To set the default connection access mode, specify the access in a CALL statement. For example:

To specify read-write access for the system administrator userID `sa` and the read-only as the default access for anyone else who connects to the database, issue these CALL statements:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'sa')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    'readOnlyAccess')
```

To specify read-write access for the user ID Fred and no access for other users, issue these CALL statements:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', 'Fred')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode',
    'noAccess')
```

Setting access for individual users

Use the `derby.database.fullAccessUsers` and `derby.database.readOnlyAccessUsers` properties to specify the user IDs that have read-write access and read-only access to a database.

You can specify multiple user IDs by using a comma-separated list, with no spaces between the comma and the next user ID.

To set the user authorizations for individual users, specify the access in a CALL statement. For example:

To specify read-write access for the system administrator user ID `sa` and for the user ID `maria`, issue this CALL statement:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
```

```
    'derby.database.fullAccessUsers', 'sa,maria')
```

To specify read-only access for a guest user ID and for Fred, issue this CALL statement:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers', 'guest,Fred')
```

To specify read-write access for the user ID "Elena!", use delimited identifiers for the user ID. For example:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers', '"Elena!"')
```

Read-only and full access permissions

The actions that users can perform on a Derby database is determined by the type of access that users have to the database objects.

The following table lists the actions that users can perform based on the type of access that a user is granted on a database. These actions apply to regular databases, source databases, and target databases.

Table 12. Actions that are authorized by type of access

Action	Read-only access	Full access
Executing SELECT statements	X	X
Reading database properties	X	X
Loading database classes from jar files	X	X
Executing INSERT, UPDATE, or DELETE statements		X
Executing DDL statements		X
Adding or replacing jar files		X
Setting database properties		X

User authorization exceptions

SQL exceptions are returned when errors occur with user authorizations.

Derby validates the database properties when you set the properties. An exception is returned if you specify an invalid value when you set these properties.

If a user attempts to connect to a database but is not authorized to connect to that database, the SQLException 04501 is returned.

If a user with read-only access attempts to write to a database, the SQLException 08004 `- connection refused` is returned.

Setting the SQL standard authorization mode

Use the `derby.database.sqlAuthorization` property to enable SQL standard authorization.

The `derby.database.sqlAuthorization` property controls the ability for object owners to grant and revoke permission for users to perform actions on database objects. It also controls the ability for users to create, set, and drop roles.

The valid settings for the `derby.database.sqlAuthorization` property are:

- *TRUE*
- *FALSE*

The default setting for the `derby.database.sqlAuthorization` property is *FALSE*.

The `derby.database.sqlAuthorization` property is usable only if the property `derby.connection.requireAuthentication` is also set to true, since SQL authorization is of no value unless authentication is also enabled.

After you set the `derby.database.sqlAuthorization` property to *TRUE*, you cannot set the property back to *FALSE*.

You can set the `derby.database.sqlAuthorization` property as a system property or as a database property. If you set this property as a system property before you create the databases, all new databases will automatically have SQL authorization enabled. If the databases already exists, you can set this property only as a database property.

To enable SQL standard authorization for the entire system, set the `derby.database.sqlAuthorization` property as a system property:

```
derby.database.sqlAuthorization=true
```

To enable SQL standard authorization for a specific database, set the `derby.database.sqlAuthorization` property as a database property:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.sqlAuthorization',
    'true')
```

Using SQL standard authorization

When the SQL standard authorization mode is enabled, object owners can use the GRANT and REVOKE SQL statements to set the user permissions for specific database objects or for specific SQL actions. They can also use roles to administer privileges.

The SQL standard authorization mode is a SQL2003 compatible access control system. You enable the SQL standard authorization mode by setting the `derby.database.sqlAuthorization` property to `TRUE`.

While Derby has a simpler database access mode which can be set to provide users with *full*, *read-only*, or *no access* authorization, this simpler access mode is less appropriate for most client-server database configurations. When users or applications issue SQL statements directly against the database, the Derby SQL authorization mode provides a more precise mechanism to limit the actions that users can take on the database.

The GRANT statement is used to grant specific permissions to users or to roles, or to grant roles to users or to roles. The REVOKE statement is used to revoke permissions and role grants. The grant and revoke privileges are:

- DELETE
- EXECUTE
- INSERT
- SELECT
- REFERENCES
- TRIGGER
- UPDATE

When a table, view, function, or procedure is created, the person that creates the object is referred to as the owner of the object. Only the object owner and the database owner have full privileges on the object. No other users have privileges on the object until the object owner grants privileges to them.

Public and individual user privileges

The object owner can grant and revoke privileges for specific users, for specific roles, or for all users. The keyword PUBLIC is used to specify all users. When PUBLIC is specified, the privileges affect all current and future users. The privileges granted and revoked to PUBLIC and to individual users or roles are independent. For example, a SELECT privilege on table `t` is granted to both PUBLIC and to the user `harry`. The SELECT privilege is later revoked from user `harry`, but user `harry` has access to table `t` through the PUBLIC privilege.

Exception: When you create a view, trigger, or constraint, Derby first checks to determine if you have the required privileges at the user level. If you have the user-level privileges, the object is created and is dependent on that user-level privilege. If you do not have the required privileges at the user-level, Derby checks to determine if you have the required privileges at the PUBLIC level. If you have the PUBLIC level privileges, the object is created and is dependent on that PUBLIC level privilege. After the object is created, if the privilege on which the object depends on is revoked, the object is automatically dropped. Derby does not try to determine if you have other privileges that can replace the privileges that are being revoked.

Example 1

User `zhi` creates table `t1` and grants SELECT privileges to user `harry` on table `t1`. User `zhi` grants SELECT privileges to PUBLIC on table `t1`. User `harry` creates view `v1` with the statement SELECT * from `zhi.t1`. The view depends on the user-level privilege that user `harry` has on `t1`. Subsequently, user `zhi` revokes SELECT privileges from user `harry` on table `t1`. As a result, the view `harry.v1` is dropped.

Example 2

User `anita` creates table `t1` and grants SELECT privileges to PUBLIC. User `harry` creates view `v1` with the statement SELECT * from `anita.t1`. The view depends on the PUBLIC level privilege that user `harry` has on `t1` since user `harry` does not have user-level privileges on table `t1` when he creates the view `harry.v1`. Subsequently, user `anita` grants SELECT privileges to user `harry` on table `anita.t1`. The view `harry.v1` continues to depend on PUBLIC level privilege that user `harry` has on `t1`. When user `anita` revokes SELECT privileges from PUBLIC on table `t1`, the view `harry.v1` is dropped.

Permissions on views, triggers, and constraints

Views, triggers, and constraints operate with the permissions of the owner of the view, trigger, or constraint. For example, user `anita` wants to create a view using the following statement:

```
CREATE VIEW s.v(vc1,vc2,vc3)
 AS SELECT t1.c1,t1.c2,f(t1.c3)
  FROM t1 JOIN t2 ON t1.c1 = t2.c1
 WHERE t2.c2 = 5
```

User `anita` needs the following permissions to create the view:

- Ownership of the schema `s`, so that she can create something in the schema
- Ownership of the table `t1`, so that she can allow others to see columns in the table
- SELECT permission on column `t2.c1` and column `t2.c2`
- EXECUTE permission on function `f`

When the view is created, only user `anita` has SELECT permission on it. User `anita` can grant SELECT permission on any or all of the columns of view `s.v` to anyone, even to users that do not have SELECT permission on `t1` or `t2`, or EXECUTE permission on `f`. User `anita` grants SELECT permission on view `s.v` to user `harry`. When user `harry` issues a SELECT statement on the view `s.v`, Derby checks to determine if user `harry` has SELECT permission on view`s.v`. Derby does not check to determine if user `harry` has SELECT permission on `t1`, or `t2`, or EXECUTE permission on `f`.

Permissions on triggers and constraints work the same way as permissions on views. When a view, trigger, or constraint is created, Derby checks that the owner has the required permissions. Other users do not need to have those permissions to perform actions on a view, trigger, or constraint.

If the required permissions are revoked from the owner of a view, trigger, or constraint, the object is dropped as part of the REVOKE statement.

See the *Java DB Reference Manual* for more information on the GRANT and REVOKE statements.

Using SQL roles

When the SQL standard authorization mode is enabled, object owners can use the SQL roles facility to administer privileges.

SQL roles are useful for administering privileges when a database has many users. Roles provide a more powerful way to grant privileges to users' sessions than to grant privileges to each user of the database, which easily becomes tedious and error-prone when many users are involved. Roles do not in and of themselves give better database security, but used correctly, they facilitate better security. Only the database owner can create, grant, revoke, and drop roles. However, object owners can grant and revoke privileges for those objects to and from roles, as well as to and from individual users and PUBLIC (all users).

Note: Derby implements a subset of SQL roles. The fact that only the database owner can create, grant, revoke, and drop roles is an implementation restriction.

Creating and granting roles

Roles are available only when SQL authorization mode is enabled (that is, when the property `derby.database.sqlAuthorization` is set to `TRUE`).

Old databases must be (hard) upgraded to at least Release 10.5 before roles can be used.

If SQL authorization mode is enabled, the database owner can use the CREATE ROLE statement to create roles. The database owner can then use the GRANT statement to grant a role to one or more users, to PUBLIC, or to another role.

A role A *contains* another role B if role B is granted to role A, or is contained in a role C granted to role A. Privileges granted to a contained role are inherited by the containing roles. So the set of privileges identified by role A is the union of the privileges granted to role A and the privileges granted to any contained roles of role A.

For example, suppose the database owner issued the following statements:

```
create role reader;
create role updater;
create role taskLeaderA;
create role taskLeaderB;
create role projectLeader;
grant reader to updater;
grant updater to taskLeaderA;
grant updater to taskLeaderB;
grant taskLeaderA to projectLeader;
grant taskLeaderB to projectLeader;
```

The roles would then have the following containment relationships:

```
                   reader
                     |
                     v
                  updater
                 /       \
```

```
       taskLeaderA       taskLeaderB
                 \       /
               projectLeader
```

In this case, the `projectLeader` role contains all the other roles and has all their privileges. If the database owner then revokes `updater` from `taskLeaderA`, `projectLeader` still contains that role through `taskLeaderB`.

The SYSCS_DIAG.CONTAINED_ROLES diagnostic table function can be used to determine the set of contained roles for a role.

Cycles are not permitted in role grants. That is, if a role contains another role, you cannot grant the container role to the contained role. For example, the following statement would not be permitted:

```
grant projectLeader to updater;
```

Setting roles

When a user first connects to Derby, no role is set, and the CURRENT_ROLE function returns null. During a session, the user can call the SET ROLE statement to set the current role for that session. The role can be any role that has been granted to the session's current user or to PUBLIC. To unset the current role, call SET ROLE with an argument of NONE. At any time during a session, there is always a current user, but there is a current role only if SET ROLE has been called with an argument other than NONE. If a current role is not set, the session has only the privileges granted to the user directly or to PUBLIC.

For example, if the database owner created and granted the roles shown in the previous session, a user would have to issue a SET ROLE statement to have them take effect. Suppose a user issued the following statement:

```
SET ROLE taskLeaderA;
```

Assuming that the database owner had granted the `taskLeaderA` role to the user, the user would be allowed to set the role as shown and would have all the privileges granted to the `taskLeaderA`, `updater`, and `reader` roles.

To retrieve the current role identifier in SQL, call the CURRENT_ROLE function.

Within stored procedures and functions that contain SQL, the current role is on the authorization stack. Initially, inside a nested connection, the current role is set to that of the calling context. Upon return from the stored procedure or function, the authorization stack is popped, so the current role of the calling context is not affected by any setting of the role inside the called procedure or function. If the stored procedure opens more than one nested connection, these all share the same (stacked) current role state. Any dynamic result set passed out of a stored procedure sees the current role of the nested context.

Granting privileges to roles

Once a role has been created, both the database owner and the object owner can grant privileges on tables and routines to that role. You can grant the same privileges to roles that you can grant to users. Granting a privilege to a role implicitly grants privileges to all roles that contain that role. For example, if you grant delete privileges on a table to `updater`, every user in the `updater`, `taskLeaderA`, `taskLeaderB`, and `projectLeader` role will also have delete privileges on that table, but users in the `reader` role will not.

Revoking privileges from a role

Either the database owner or the object owner can revoke privileges from a role.

When a privilege is revoked from a role A, that privilege is no longer held by role A, unless A otherwise inherits that privilege from a contained role.

If a privilege to an object is revoked from role A, a session will lose that privilege if it has a current role set to A or a role that contains A, unless one or more of the following is true:

- The privilege is granted directly to the current user
- The privilege is granted to PUBLIC
- The privilege is also granted to another role B in the current role's set of contained roles
- The session's current user is the database owner or the object owner

Revoking roles

The database owner can use the REVOKE statement to revoke a role from a user, from PUBLIC, or from another role.

When a role is revoked from a user, that session can no longer keep that role, nor can it take on that role in a SET ROLE statement, unless the role is also granted to PUBLIC. If that role is the current role of an existing session, the current privileges of the session lose any extra privileges obtained through setting that role.

The default drop behavior is CASCADE. Therefore, all persistent objects (constraints, views and triggers) that rely on that role are dropped. Although there may be other ways of fulfilling that privilege at the time of the revoke, any dependent objects are still dropped. This is an implementation limitation. Any prepared statement that is potentially affected will be checked again on the next execute. A result set that depends on a role will remain open even if that role is revoked from a user.

When a role is revoked from a role, the default drop behavior is also CASCADE. Suppose you revoke role A from role B. Revoking the role will have the effect of revoking all additional applicable privileges obtained through A from B. Roles that contain B will also lose those privileges, unless A is still contained in some other role C granted to B, or the privileges come through some other role. See Creating and granting roles for an example.

Dropping roles

Only the database owner can drop a role. To drop a role, use the DROP ROLE statement.

Dropping a role effectively revokes all grants of this role to users and other roles.

Further information

For details on the following statements, functions, and system table related to roles, see the *Java DB Reference Manual*.

- CREATE ROLE statement
- SET ROLE statement
- DROP ROLE statement
- GRANT statement
- REVOKE statement
- CURRENT_ROLE function
- SYSCS_DIAG.CONTAINED_ROLES table function
- SYSROLES system table

SQL standard authorization exceptions

SQL exceptions are returned when errors occur with SQL authorization.

The following errors can result from the CREATE ROLE statement:

- You cannot create a role if you are not the database owner. An attempt to do so raises the *SQLException* 4251A.
- You cannot create a role if a role with that name already exists. An attempt to do so raises the *SQLException* X0Y68.
- You cannot create a role name if there is a user by that name. An attempt to create a role name that conflicts with an existing user name raises the *SQLException* X0Y68.
- A role name cannot start with the prefix SYS (after case normalization). Use of the prefix SYS raises the *SQLException* 4293A.
- You cannot create a role with the name PUBLIC (after case normalization). PUBLIC is a reserved authorization identifier. An attempt to create a role with the name PUBLIC raises *SQLException* 4251B.

The following errors can result from the DROP ROLE statement:

- You cannot drop a role if you are not the database owner. An attempt to do so raises the *SQLException* 4251A.
- You cannot drop a role that does not exist. An attempt to do so raises the *SQLException* 0P000.

The following errors can result from the SET ROLE statement:

- You cannot set a role if you are not the database owner. An attempt to do so raises the *SQLException* 4251A.
- You cannot set a role that does not exist. An attempt to do so raises the *SQLException* 0P000.
- You cannot set a role when a transaction is in progress. An attempt to do so raises the *SQLException* 25001.
- You cannot use NONE or a malformed identifier as a string or ? argument to SET ROLE. An attempt to do so raises the *SQLException* XCXA0.

The following errors can result from the GRANT statement:

- You cannot grant a role if you are not the database owner. An attempt to do so raises the *SQLException* 4251A.
- You cannot grant a role that does not exist. An attempt to do so raises the *SQLException* 0P000.
- You cannot grant the role "PUBLIC". An attempt to do so raises the *SQLException* 4251B.
- You cannot grant a role if doing so would create a circularity by granting a container role to a contained role. An attempt to do so raises the *SQLException* 4251C.

The following errors can result from the REVOKE statement:

- You cannot revoke a role if you are not the database owner. An attempt to do so raises the *SQLException* 4251A.
- You cannot revoke a role that does not exist. An attempt to do so raises the *SQLException* 0P000.
- You cannot revoke the role "PUBLIC". An attempt to do so raises the *SQLException* 4251B.

For all statements, an attempt to specify an identifier argument more than 128 characters long raises the *SQLException* 42622.

Encrypting databases on disk

Derby provides a way for you to encrypt your data on disk.

Typically, database systems encrypt and decrypt data in transport over the network, using industry-standard systems. This system works well for client/server databases;

the server is assumed to be in a trusted, safe environment, managed by a system administrator. In addition, the recipient of the data is trusted and should be capable of protecting the data. The only risk comes when transporting data over the wire, and data encryption happens during network transport only.

However, Derby databases are platform-independent files that are designed to be easily shared in a number of ways, including transport over the Internet. Recipients of the data might not know how, or might not have the means, to properly protect the data.

This data encryption feature provides the ability to store user data in an encrypted form. The user who boots the database must provide a boot password.

Note: Jar files stored in the database are not encrypted.

Requirements for Derby encryption

Derby supports disk encryption and requires an encryption provider. An encryption provider implements the Java cryptography concepts. The JRE for J2SE 1.4 and higher includes Java Cryptographic Extensions (JCE http://java.sun.com/products/jce/index.html) and one or more default encryption providers.

Working with encryption

This section describes using encryption in Derby.

Encrypting databases on creation

You configure a Derby database for encryption when you create the database by specifying the *dataEncryption=true* attribute on the connection URL.

The Java Runtime Environment (JRE) determines the default encryption provider, as follows:

- For J2SE/J2EE 1.4 or higher, the JRE's provider is the default.
- If your environment for some reason does not include a provider, it must be specified.

You have the option of specifying an alternate encryption provider. The default encryption algorithm is DES, but you have the option of specifying an alternate algorithm as well. See Specifying an alternate encryption provider

Encrypting an existing unencrypted database

You can encrypt an unencrypted Derby database by specifying attributes on the connection URL when you boot the database. The attributes that you specify depend on how you want the database encrypted.

- If the database is configured with log archival, you must disable log archival and perform a shutdown before you can encrypt the database.
- If there are any global transaction that are in the prepared state after recovery, the database cannot be encrypted.

When you encrypt an existing, unencrypted database, you can specify whether the database should be encrypted using a boot password or an external encryption key. You can also specify the *encryptionProvider* attribute and the *encryptionAlgorithm* attribute on the connection URL. The database is configure with the specified encryption attributes and all of the existing data in the database is encrypted.

Encrypting a database is a time consuming process because it involves encrypting all of the existing data in the database. If the process is interrupted before completion, all the changes are rolled back the next time that the database is booted. If the interruption

occurs immediately after the database is encrypted but before the connection is returned to the application, you might not be able to boot the database without the boot password or external encryption key. In these rare circumstances, you should try to boot the database with the boot password or the external encryption key.

Recommendation: Ensure that you have enough free disk space before you encrypt a database. In addition to the disk space required for the current size of the database, temporary disk space is required to store the old version of the data to restore the database back to it's original state if the encryption is interrupted or returns errors. All of the temporary disk space is released back to the operating system after the database is encrypted.

To encrypt an existing unencrypted database:

1. Specify the *dataEncryption=true* attribute and either the *encryptionKey* attribute or the *bootPassword* attribute in a URL and boot the database.

   For example, to encrypt the `salesdb` database with the boot password `abc1234xyz`, specify the following attributes in the URL:

   ```
   jdbc:derby:salesdb;dataEncryption=true;bootPassword=abc1234xyz
   ```

   If authentication and SQL authorization are both enabled, the credentials of the database owner must be supplied as well, since encryption is a restricted operation.

   If you disabled log archival before you encrypted the database, create a new backup of the database after the database is encrypted.

Creating the boot password

When you encrypt a database you must also specify a boot password, which is an alpha-numeric string used to generate the encryption key.

The length of the encryption key depends on the algorithm used:

- AES (128, 192, and 256 bits)
- DES (the default) (56 bits)
- DESede (168 bits)
- All other algorithms (128 bits)

Note: The boot password should have at least as many characters as number of bytes in the encryption key (56 bits=8 bytes, 168 bits=24 bytes, 128 bits=16 bytes). The minimum number of characters for the boot password allowed by Derby is eight.

It is a good idea not to use words that would be easily guessed, such as a login name or simple words or numbers. A *bootPassword*, like any password, should be a mix of numbers and upper- and lowercase letters.

You turn on and configure encryption and specify the corresponding boot password on the connection URL for a database when you create it:

```
jdbc:derby:encryptionDB1;create=true;dataEncryption=true;
    bootPassword=clo760uds2caPe
```

Note: If you lose the *bootPassword* and the database is not currently booted, you will not be able to connect to the database anymore. (If you know the current *bootPassword*, you can change it. See Encrypting databases with a new key.)

Specifying an alternate encryption provider:

You can specify an alternate provider when you create the database with the *`encryptionProvider=providerName`* attribute.

You must specify the full package and class name of the provider, and you must also add the libraries to the application's classpath.

```
-- using the the provider library jce_jdk13-10b4.zip|
-- available from www.bouncycastle.org
jdbc:derby:encryptedDB3;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe;
encryptionProvider=org.bouncycastle.jce.provider.BouncyCastleProvider;
encryptionAlgorithm=DES/CBC/NoPadding

-- using a provider
-- available from
-- http://jcewww.iaik.tu-graz.ac.at/download.html
jdbc:derby:encryptedDB3;create=true;dataEncryption=true;
bootPassword=clo760uds2caPe;
encryptionProvider=iaik.security.provider.IAIK;encryptionAlgorithm=
DES/CBC/NoPadding
```

Specifying an alternate encryption algorithm:

Derby supports the following encryption algorithms.

- DES (the default)
- DESede (also known as triple DES)
- Any encryption algorithm that fulfills the following requirements:
    - It is symmetric
    - It is a block cipher, with a block size of 8 bytes
    - It uses the *NoPadding* padding scheme
    - Its secret key can be represented as an arbitrary byte array
    - It requires exactly one initialization parameter, an initialization vector of type *javax.crypto.spec.IvParameterSpec*
    - It can use *javax.crypto.spec.SecretKeySpec* to represent its key

  For example, the algorithm *Blowfish* implemented in the Sun JCE package fulfills these requirements.

By Java convention, an encryption algorithm is specified like this:

```
algorithmName/feedbackMode/padding
```

The only feedback modes allowed are:

- CBC
- CFB
- ECB
- OFB

The only padding mode allowed is *NoPadding*.

By default, Derby uses the DES algorithm of *DES/CBC/NoPadding*.

To specify an alternate encryption algorithm when you create a database, use the *encryptionAlgorithm=algorithm* attribute. If the algorithm you specify is not supported by the provider you have specified, Derby throws an exception.

To specify the AES encryption algorithm with a key length other than the default of 128, specify the *encryptionKeyLength* attribute. For example, you might specify the following connection attributes:

```
jdbc:derby:encdbcbc_192;create=true;dataEncryption=true;
encryptionKeyLength=192;encryptionAlgorithm=AES/CBC/NoPadding;
bootPassword=Thursday
```

To use the AES algorithm with a key length of 192 or 256, you must use unrestricted policy jar files for your JRE. You can obtain these files from your Java provider. They might have a name like "Java Cryptography Extension (JCE) Unlimited Strength

Jurisdiction Policy Files." If you specify a non-default key length using the default policy jar files, a Java exception occurs.

Encrypting databases with a new key

You can apply a new encryption key to a Derby database by specifying a new boot password or a new external key.

Encrypting a database with a new encryption key is a time consuming process because it involves encrypting all of the existing data in the database with the new encryption key. If the process is interrupted before completion, all the changes are rolled back the next time that the database is booted. If the interruption occurs immediately after the database is encrypted with the new encryption key but before the connection is returned to the application, you might not be able to boot the database with the old encryption key. In these rare circumstances, you should try to boot the database with the new encryption key.

Recommendation: Ensure that you have enough free disk space before you encrypt a database with a new key. In addition to the disk space required for the current size of the database, temporary disk space is required to store the old version of the data to restore the database back to it's original state if the new encryption is interrupted or returns errors. All of the temporary disk space is released back to the operating system after the database is reconfigured to work with the new encryption key.

To encrypt a database with a new encryption key:

1. Use the type of encryption that is currently used to encrypt the database:
   - To encrypt the database with a new boot password key, use the *newBootPassword* attribute.
   - To encrypt the database with a new external encryption key, use the *newEncryptionKey* attribute.

   If authentication and SQL authorization are both enabled, the credentials of the database owner must be supplied, since reencryption is a restricted operation.

Encrypting databases with a new boot password:

You can apply a new boot password to a Derby database by specifying the *newBootPassword* attribute on the connection URL when you boot the database.

- If the database is configured with log archival for roll-forward recovery, you must disable log archival and perform a shutdown before you can encrypt the database with a new boot password.
- If there are any global transaction that are in the prepared state after recovery, the database cannot be encrypted with a new boot password.
- If the database is currently encrypted with an external encryption key, you should use the *newEncryptionKey* attribute to encrypt the database.

When you use the *newBootPassword* attribute, a new encryption key is generated internally by the engine and the key is protected using the new boot password. The newly generated encryption key encrypts the database, including the existing data. You cannot change the encryption provider or encryption algorithm when you apply a new boot password.

To encrypt a database with a new boot password:

1. Specify the *newBootPassword* attribute in a URL and reboot the database.

   For example, when the following URL is used when the `salesdb` database is rebooted, the database is encrypted with the new encryption key, and is protected by the password new1234xyz:

   ```
   jdbc:derby:salesdb;bootPassword=abc1234xyz;newBootPassword=new1234xyz
   ```

If authentication and SQL authorization are both enabled, the credentials of the database owner must be supplied as well, since reencryption is a restricted operation.

If you disabled log archival before you applied the new boot password, create a new backup of the database after the database is reconfigured with the new boot password.

Encrypting databases with a new external encryption key:

You can apply a new external encryption key to a Derby database by specifying the *newEncryptionKey* attribute on the connection URL when you boot the database.

- If the database is configured with log archival for roll-forward recovery, you must disable log archival and perform a shutdown before you can encrypt the database with a new external encryption key.
- If there are any global transaction that are in the prepared state after recovery, the database cannot be encrypted with a new encryption key.
- If the database is currently encrypted with a boot password , you should use the *newBootPassword* attribute to encrypt the database.

To encrypt a database with a new external encryption key:

1. Specify the *newEncryptionKey* attribute in a URL and reboot the database.

   For example, when the following URL is used when the `salesdb` database is rebooted, the database is encrypted with the new encryption key 6862636465666768:

   ```
   jdbc:derby:salesdb;encryptionKey=6162636465666768;newEncryptionKey=6862636465666768
   ```

   If authentication and SQL authorization are both enabled, the credentials of the database owner must be supplied as well, since encryption is a restricted operation.

   If you disabled log archival before you applied the new encryption key, create a new backup of the database after the database is reconfigured with new the encryption key.

Booting an encrypted database

If you create an encrypted database using the *bootPassword* attribute, you must specify the boot password to reboot the database. If you create an encrypted database using the *encryptionKey* attribute, you must specify the *encryptionKey* to reboot the database.

Encrypted databases cannot be booted automatically along with all other system databases on system startup (see "*derby.system.bootAll*" in the *Java DB Reference Manual*). Instead, you boot encrypted databases when you first connect to the database.

Booting a database with the *bootPassword* attribute

To access an encrypted database called `wombat` that was created with the boot password `clo760uds2caPe`, use the following connection URL:

```
jdbc:derby:wombat;bootPassword=clo760uds2caPe
```

Booting a database with the *encryptionKey* attribute

To access an encrypted database called `flintstone` that was created with the `encryptionKey=c566bab9ee8b62a5ddb4d9229224c678` and with the `encryptionAlgorithm=AES/CBC/NoPadding`, use the following connection URL:

```
jdbc:derby:flintstone;encryptionAlgorithm=AES/CBC/NoPadding;
encryptionKey=c566bab9ee8b62a5ddb4d9229224c678
```

After the database is booted, all connections can access the database without the boot password. Only a connection that boots the database requires the key.

For example, the following connections would boot the database and require the boot password or encryption key, depending on what mechanism was used to encrypt the database originally:

- The first connection to the database in the JVM session
- The first connection to the database after the database has been explicitly shut down
- The first connection to the database after the system has been shut down and then rebooted

Note: The boot password and the encryption key are not meant to prevent unauthorized connections to the database after the database is booted. To protect a database after it has been booted, turn on user authentication (see Working with user authentication).

Signed jar files

In a Java 2 environment, Derby can detect digital signatures on jar files. When attempting to load a class from a signed jar file stored in the database, Derby will verify the validity of the signature.

Note: The Derby class loader only validates the integrity of the signed jar file and that the certificate has not expired. Derby cannot ascertain whether the validity/identity of declared signer is correct. To validate identity, use a Security Manager (i.e., an implementation of *java.lang.SecurityManager*).

When loading classes from an application jar file in a Java 2 environment, Derby behaves as follows:

- *If the class is signed, Derby will:*
  - Verify that the jar was signed using a X.509 certificate (i.e., can be represented by the class *java.security.cert.X509Certificate*). If not, throw an exception.
  - Verify that the digital signature matches the contents of the file. If not, throw an exception.
  - Check that the set of signing certificates are all valid for the current date and time. If any certificate has expired or is not yet valid, throw an exception.
  - Pass the array of certificates to the *setSigners()* method of `java.lang.ClassLoader`. This allows security managers to obtain the list of signers for a class (using `java.lang.Class.getSigners`) and then validate the identity of the signers using the services of a Public Key Infrastructure (PKI).

Note: Derby does not provide a security manager.

For more information about signed jar files, see the Java 2 specifications at *http://java.sun.com*.

For more information about Java 2 security, go to *http://java.sun.com/security/*.

Notes on the Derby security features

TheDerby security model has some basic limitations.

You lock out non full-access users with database properties, which are stored in the database (and in an encrypted database these properties are also encrypted). Note, however, for a distributed/embedded system that a sophisticated user with the database encryption key might be able to physically change those properties in the database files.

In addition, in the Derby system, it is not necessary to have a specific connection (or permission to access a particular database) to shut down the system. Any authenticated user can shut down the system.

Other security holes to think about are:

- JVM subversion, running the application under a home-grown JVM.
- Trolling for objects
- Class substitution, locating a class that has access to sensitive data and replacing it with one that passes on information

User authentication and authorization examples

This section provides examples on using user authentication and authorization in Derby in either a client/server environment or in an embedded environment.

User authentication example in a client/server environment

In this example, Derby is running in a user-designed application server.

Derby provides the user authentication, not the application server. The server is running in a secure environment, the application server encrypts the passwords, and a database administrator is available. The administrator configures security using system-level properties in the *derby.properties* file and has protected this file with operating system tools. Derby connects to an existing LDAP directory service within the enterprise to authenticate users.

The default access mode for all databases is set to *fullAccess* (the default).

The *derby.properties* file for the server includes the following entries:

```
# turn on user authentication
derby.connection.requireAuthentication=true
# set the authentication provider to an external LDAP server
derby.authentication.provider=LDAP
# the host name and port number of the LDAP server
derby.authentication.server=godfrey:389
# the search base for user names
derby.authentication.ldap.searchBase=o=oakland.mycompany.com
# explicitly show the access mode for databases (this is default)
derby.database.defaultConnectionMode=fullAccess
```

With these settings, all users must be authenticated by the LDAP server in order to access any Derby databases.

The database administrator has determined that one database, *accountingDB*, has additional security needs. Within a connection to that database, the database administrator uses database-wide properties (which override properties set in the *derby.properties* file) to limit access to this database. Only the users *prez*, *cfo*, and *numberCruncher* have full (read-write) access to this database, and only *clerk1* and *clerk2* have read-only access to this database. No other users can access the database.

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.defaultConnectionMode', 'noAccess')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.fullAccessUsers',
    'prez,cfo,numberCruncher')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.readOnlyAccessUsers', 'clerk1,clerk2')
```

The database administrator then requires all current users to disconnect and re-connect. These property changes do not go into effect for current connections. The database administrator can force current users to reconnect by shutting down the database

User authentication and authorization client example

This example consists of a pair of programs, `AuthExampleClient1.java` and `AuthExampleClient2.java`, which show how to turn on, use, and turn off user authentication using Derby's built-in user authentication and user authorization using the client driver.

This example uses one program to set properties and a second program to perform database operations. A similar example that uses the embedded driver, in User authentication and authorization embedded example, is a single long program. Either example would work equally well in the other format.

See User authentication and SQL authorization client example for an example similar to this one that uses SQL authorization.

The first program, `AuthExampleClient1.java`, does the following:

1. Creates a database named `authClientDB`, using the client driver.
2. Sets database properties that create users with different levels of access (read-only and full access), require authentication, and set the default access level to no access.
3. Closes the connection and shuts down the database.

The second program, `AuthExampleClient2.java`, does the following:

1. Tries to connect to the database without a username and password, raising an exception.
2. Connects to the database as a user with read-only access; the connection succeeds, but an attempt to create a table raises an exception.
3. Connects to the database as a user with full access; this user can create and populate a table.
4. Removes the table.
5. Closes the connection and shuts down the database.

Make sure that the `javac` command is in your path, then compile the programs as follows:

```
javac AuthExampleClient1.java
```

```
javac AuthExampleClient2.java
```

Before you run the programs, start the Derby Network Server as described in step 2 of "Activity 4: Create and run a JDBC program using the client driver and Network Server" in *Getting Started with Java DB*. When you run the programs, make sure that `%DERBY_HOME%\lib\derbyclient.jar` (or `$DERBY_HOME/lib/derbyclient.jar`) is in your classpath. For example, you might use the following commands on a Windows system:

```
java -cp .;%DERBY_HOME%\lib\derbyclient.jar AuthExampleClient1
```

```
java -cp .;%DERBY_HOME%\lib\derbyclient.jar AuthExampleClient2
```

Source code for `AuthExampleClient1.java`

```
import java.sql.*;

public class AuthExampleClient1 {

    public static void main(String[] args) {

        String driver = "org.apache.derby.jdbc.ClientDriver";
        String dbName="authClientDB";
        String connectionURL = "jdbc:derby://localhost:1527/" + dbName +
            ";create=true";
        Connection conn = null;
```

```
        // Load the driver. This code is not needed if you are using
        // JDK 6, because in that environment the driver is loaded
        // automatically when the application requests a connection.
        try {
            Class.forName(driver);
            System.out.println(driver + " loaded.");
        } catch (java.lang.ClassNotFoundException ce) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(ce.getMessage());
            System.out.println("\n Make sure your CLASSPATH variable " +
                "contains %DERBY_HOME%\\lib\\derbyclient.jar " +
                "(${DERBY_HOME}/lib/derbyclient.jar).\n");
        } catch (Exception ee) {
            errorPrintAndExit(ee);
        }

        // Create and boot the database and set up users, then shut down
        // the database as one of the users with full access
        try {
            System.out.println("Trying to connect to " + connectionURL);
            conn = DriverManager.getConnection(connectionURL);
            System.out.println("Connected to database " + connectionURL);

            turnOnBuiltInUsers(conn);

            // Close connection
            conn.close();
            System.out.println("Closed connection");

            /* Shut down the database to make static properties take
             * effect. Because the default connection mode is now
             * noAccess, you must specify a user that has access. But
             * because requireAuthentication does not take effect until
             * you restart the database, the password is not checked.
             *
             * Database shutdown throws the 08006 exception to confirm
             * success.
             */
            try {
                DriverManager.getConnection(
                    "jdbc:derby://localhost:1527/" + dbName +
                    ";user=sa;password=badpass;shutdown=true");
            } catch (SQLException se)  {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");
        } catch (Throwable e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Turn on built-in user authentication and user authorization.
     *
     * @param conn a connection to the database.
     */
    public static void turnOnBuiltInUsers(Connection conn)
            throws SQLException {

        String setProperty =
            "CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(";
        String getProperty =
            "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(";
        String requireAuth = "'derby.connection.requireAuthentication'";
        String defaultConnMode =
            "'derby.database.defaultConnectionMode'";
        String fullAccessUsers = "'derby.database.fullAccessUsers'";
```

```
        String readOnlyAccessUsers =
            "'derby.database.readOnlyAccessUsers'";
        String provider = "'derby.authentication.provider'";
        String propertiesOnly = "'derby.database.propertiesOnly'";

        System.out.println("Turning on authentication.");
        Statement s = conn.createStatement();

        // Set and confirm requireAuthentication
        s.executeUpdate(setProperty + requireAuth + ", 'true')");
        ResultSet rs = s.executeQuery(getProperty + requireAuth + ")");
        rs.next();
        System.out.println("Value of requireAuthentication is " +
            rs.getString(1));

        // Set authentication scheme to Derby builtin
        s.executeUpdate(setProperty + provider + ", 'BUILTIN')");

        // Create some sample users
        s.executeUpdate(
            setProperty + "'derby.user.sa', 'ajaxj3x9')");
        s.executeUpdate(
            setProperty + "'derby.user.guest', 'java5w6x')");
        s.executeUpdate(
            setProperty + "'derby.user.mary', 'little7xylamb')");

        // Define noAccess as default connection mode
        s.executeUpdate(
            setProperty + defaultConnMode + ", 'noAccess')");

        // Confirm default connection mode
        rs = s.executeQuery(getProperty + defaultConnMode + ")");
        rs.next();
        System.out.println("Value of defaultConnectionMode is " +
            rs.getString(1));

        // Define read-write users
        s.executeUpdate(
            setProperty + fullAccessUsers + ", 'sa,mary')");

        // Define read-only user
        s.executeUpdate(
            setProperty + readOnlyAccessUsers + ", 'guest')");

        // Confirm full-access users
        rs = s.executeQuery(getProperty + fullAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of fullAccessUsers is " + rs.getString(1));

        // Confirm read-only users
        rs = s.executeQuery(getProperty + readOnlyAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of readOnlyAccessUsers is " + rs.getString(1));

        // We would set the following property to TRUE only when we were
        // ready to deploy. Setting it to FALSE means that we can always
        // override using system properties if we accidentally paint
        // ourselves into a corner.
        s.executeUpdate("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
            "'derby.database.propertiesOnly', 'false')");
        s.close();
    }

    /**
     * Report exceptions, with special handling of SQLExceptions,
     * and exit.
     *
     * @param e an exception (Throwable)
```

```
     */
    static void errorPrintAndExit(Throwable e) {
        if (e instanceof SQLException)
            SQLExceptionPrint((SQLException)e);
        else {
            System.out.println("A non-SQL error occurred.");
            e.printStackTrace();
        }
        System.exit(1);
    }

    /**
     * Iterate through a stack of SQLExceptions.
     *
     * @param sqle a SQLException
     */
    static void SQLExceptionPrint(SQLException sqle) {
        while (sqle != null) {
            System.out.println("\n---SQLException Caught---\n");
            System.out.println("SQLState:   " + (sqle).getSQLState());
            System.out.println("Severity: " + (sqle).getErrorCode());
            System.out.println("Message:  " + (sqle).getMessage());
            sqle = sqle.getNextException();
        }
    }
}
```

Source code for `AuthExampleClient2.java`

```
import java.sql.*;

public class AuthExampleClient2 {

    public static void main(String[] args) {

        String driver = "org.apache.derby.jdbc.ClientDriver";
        String dbName="authClientDB";
        String connectionURL = "jdbc:derby://localhost:1527/" + dbName;
        Connection conn = null;

        // Restart database and confirm that unauthorized users cannot
        //  access it

        // Load the driver. This code is not needed if you are using
        // JDK 6, because in that environment the driver is loaded
        // automatically when the application requests a connection.
        try {
            Class.forName(driver);
            System.out.println(driver + " loaded.");
        } catch (java.lang.ClassNotFoundException ce) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(ce.getMessage());
            System.out.println("\n Make sure your CLASSPATH variable " +
                "contains %DERBY_HOME%\\lib\\derbyclient.jar " +
                "(${DERBY_HOME}/lib/derbyclient.jar). \n");
        } catch (Exception ee) {
            errorPrintAndExit(ee);
        }

        // Try to log in with no username or password
        try {
            // connection attempt should fail
            System.out.println("Trying to connect to " + connectionURL +
                " without username or password");
            conn = DriverManager.getConnection(connectionURL);
            System.out.println(
                "ERROR: Unexpectedly connected to database " + dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("08004")) {
```

```
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with read-only access
        try {
            // connection should succeed, but create table should fail
            String newURL = connectionURL +
                ";user=guest;password=java5w6x";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName +
                " with read-only access");

            Statement s = conn.createStatement();
            s.executeUpdate("CREATE TABLE t1(C1 VARCHAR(6))");
            System.out.println(
                "ERROR: Unexpectedly allowed to modify database " +
                     dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("25503")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
                try {
                    conn.close();
                } catch (SQLException ee) {
                    errorPrintAndExit(ee);
                }
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with full access
        // Create, update, and query table
        try {
            // this should succeed
            String newURL = connectionURL +
                ";user=mary;password=little7xylamb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            Statement s = conn.createStatement();
            s.executeUpdate("CREATE TABLE T1(C1 VARCHAR(6))");
            System.out.println("Created table T1");
            s.executeUpdate("INSERT INTO T1 VALUES('hello')");

            ResultSet rs = s.executeQuery("SELECT * FROM T1");
            rs.next();
            System.out.println("Value of T1/C1 is " + rs.getString(1));
            s.executeUpdate("DROP TABLE T1");

            s.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        try {
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
```

```
     * Close connection and shut down database.
     *
     * @param conn a connection to the database
     */
    public static void cleanUpAndShutDown (Connection conn)
            throws SQLException {

        String dbName="authClientDB";
        String connectionURL = "jdbc:derby://localhost:1527/" + dbName;

        try {
            conn.close();
            System.out.println("Closed connection");

            // As mary, shut down the database.
            try {
                String newURL = connectionURL +
                    ";user=mary;password=little7xylamb;shutdown=true";
                DriverManager.getConnection(newURL);
            } catch (SQLException se)  {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Report exceptions, with special handling of SQLExceptions,
     * and exit.
     *
     * @param e an exception (Throwable)
     */
    static void errorPrintAndExit(Throwable e) {
        if (e instanceof SQLException)
            SQLExceptionPrint((SQLException)e);
        else {
            System.out.println("A non-SQL error occurred.");
            e.printStackTrace();
        }
        System.exit(1);
    }

    /**
     * Iterate through a stack of SQLExceptions.
     *
     * @param sqle a SQLException
     */
    static void SQLExceptionPrint(SQLException sqle) {
        while (sqle != null) {
            System.out.println("\n---SQLException Caught---\n");
            System.out.println("SQLState:   " + (sqle).getSQLState());
            System.out.println("Severity: " + (sqle).getErrorCode());
            System.out.println("Message:  " + (sqle).getMessage());
            sqle = sqle.getNextException();
        }
    }
}
```

User authentication example in a single-user, embedded environment

In this example, Derby is embedded in a single-user application that is deployed in a number of different and potentially insecure ways.

For that reason, the application developer has decided to encrypt the database and to turn on user authentication using Derby's built-in user authentication, which will not require connections to an LDAP server. The end-user must know the *bootPassword* to boot the database and the user name and password to connect to the database. Even if the database ended up in an e-mail, only the intended recipient would be able to access data in the database. The application developer has decided not to use any user authorization features, since each database will accept only a single user. In that situation, the default full-access connection mode is acceptable.

When creating the database, the application developer encrypts the database by using the following connection URL:

```
jdbc:derby:wombat;create=true;dataEncryption=true;
    bootPassword=sxy90W348HHn;user=redbaron
```

Before deploying the database, the application developer turns on user authentication, sets the authentication provider to BUILTIN, creates a single user and password, and disallows system-wide properties to protect the database-wide security property settings:

```
CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.connection.requireAuthentication', 'true')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.authentication.provider', 'BUILTIN')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.user.redbaron', 'red29PlaNe')

CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(
    'derby.database.propertiesOnly', true')
```

When the user connects (and boots) the database, the user has to provide the *bootPassword*, the user name, and the password.

Note: The user name (the value specified by the `derby.user.`*`enduser`* property) must be supplied when the database is created, even if authentication is not yet enabled. Otherwise the database owner will have the default name "APP" (see Database owner for details).

The following example shows how to provide these properties in a connection URL, although the application programmer would probably provide GUI windows to allow the end user to type those in:

```
jdbc:derby:wombat;bootPassword=sxy90W348HHn;
    user=redbaron;password=red29PlaNe
```

User authentication and authorization embedded example

This example, `AuthExampleEmbedded.java`, shows how to turn on, use, and turn off user authentication using Derby's built-in user authentication and user authorization using the embedded driver.

This example is a single long program. A similar example that uses the client driver, in User authentication and authorization client example, uses one program to set properties and a second program to perform database operations. Either example would work equally well in the other format.

See User authentication and SQL authorization embedded example for an example similar to this one that uses SQL authorization.

The program does the following:

1. Starts Derby and creates a database named `authEmbDB`, using the embedded driver.

2. Sets database properties that create users with different levels of access (read-only and full access), require authentication, and set the default access level to no access.
3. Closes the connection, then stops and restarts the database so that the authentication changes can take effect.
4. Tries to connect to the database without a username and password, raising an exception.
5. Connects to the database as a user with read-only access; the connection succeeds, but an attempt to create a table raises an exception.
6. Connects to the database as a user with full access; this user can create and populate a table.
7. Deletes the table.
8. Closes the connection, shuts down the database, then shuts down Derby.

Make sure that the `javac` command is in your path, then compile the program as follows:

```
javac AuthExampleEmbedded.java
```

When you run `AuthExampleEmbedded`, make sure that `%DERBY_HOME%\lib\derby.jar` (or `$DERBY_HOME/lib/derby.jar`) is in your classpath. For example, you might use the following command on a UNIX system:

```
java -cp .:${DERBY_HOME}/lib/derby.jar AuthExampleEmbedded
```

Source code for `AuthExampleEmbedded.java`

```
import java.sql.*;

public class AuthExampleEmbedded {

    public static void main(String[] args) {

        String driver = "org.apache.derby.jdbc.EmbeddedDriver";
        String dbName="authEmbDB";
        String connectionURL = "jdbc:derby:" + dbName + ";create=true";
        Connection conn = null;

        // Load the driver. This code is not needed if you are using
        // JDK 6, because in that environment the driver is loaded
        // automatically when the application requests a connection.
        try {
            Class.forName(driver);
            System.out.println(driver + " loaded.");
        } catch (java.lang.ClassNotFoundException ce) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(ce.getMessage());
            System.out.println("\n Make sure your CLASSPATH variable " +
                "contains %DERBY_HOME%\\lib\\derby.jar " +
                "(${DERBY_HOME}/lib/derby.jar).\n");
        } catch (Exception ee) {
            errorPrintAndExit(ee);
        }

        // Create and boot the database and set up users, then shut down
        // the database as one of the users with full access
        try {
            System.out.println("Trying to connect to " + connectionURL);
            conn = DriverManager.getConnection(connectionURL);
            System.out.println("Connected to database " + connectionURL);

            turnOnBuiltInUsers(conn);

            // close the connection
            conn.close();
            System.out.println("Closed connection");

            /* Shut down the database to make static properties take
```

```
             * effect. Because the default connection mode is now
             * noAccess, you must specify a user that has access. But
             * because requireAuthentication does not take effect until
             * you restart the database, the password is not checked.
             *
             * Database shutdown throws the 08006 exception to confirm
             * success.
             */
            try {
                DriverManager.getConnection("jdbc:derby:" + dbName +
                    ";user=sa;password=badpass;shutdown=true");
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        // Restart database and confirm that unauthorized users cannot
        //  access it
        connectionURL = "jdbc:derby:" + dbName;

        // Try to log in with no username or password
        try {
            // connection attempt should fail
            System.out.println("Trying to connect to " + connectionURL +
                " without username or password");
            conn = DriverManager.getConnection(connectionURL);
            System.out.println(
                "ERROR: Unexpectedly connected to database " + dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("08004")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with read-only access
        try {
            // connection should succeed, but create table should fail
            String newURL = connectionURL +
                ";user=guest;password=java5w6x";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName +
                " with read-only access");

            Statement s = conn.createStatement();
            s.executeUpdate("CREATE TABLE t1(C1 VARCHAR(6))");
            System.out.println(
                "ERROR: Unexpectedly allowed to modify database " +
                     dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("25503")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
                try {
                    conn.close();
                } catch (SQLException ee) {
                    errorPrintAndExit(ee);
                }
            } else {
                errorPrintAndExit(e);
```

```
            }
        }

        // Log in as a user with full access
        // Create, update, and query table
        try {
            // this should succeed
            String newURL = connectionURL +
                ";user=mary;password=little7xylamb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            Statement s = conn.createStatement();
            s.executeUpdate("CREATE TABLE T1(C1 VARCHAR(6))");
            System.out.println("Created table T1");
            s.executeUpdate("INSERT INTO T1 VALUES('hello')");

            ResultSet rs = s.executeQuery("SELECT * FROM T1");
            rs.next();
            System.out.println("Value of T1/C1 is " + rs.getString(1));
            s.executeUpdate("DROP TABLE T1");

            s.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        try {
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Close connection and shut down database. Since this is embedded
     * mode, we must also shut down the Derby system.
     *
     * @param conn a connection to the database
     */
    public static void cleanUpAndShutDown (Connection conn)
            throws SQLException {

        String dbName="authEmbDB";
        String connectionURL = "jdbc:derby:" + dbName;

        try {
            conn.close();
            System.out.println("Closed connection");

            // As mary, shut down the database.
            try {
                String newURL = connectionURL +
                    ";user=mary;password=little7xylamb;shutdown=true";
                DriverManager.getConnection(newURL);
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");

            try {
                DriverManager.getConnection("jdbc:derby:;shutdown=true");
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("XJ015") ) {
                    throw se;
                }
            }
```

```
            System.out.println("Derby system shut down normally");
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Turn on built-in user authentication and user authorization.
     *
     * @param conn a connection to the database
     */
    public static void turnOnBuiltInUsers(Connection conn)
            throws SQLException {

        String setProperty =
            "CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(";
        String getProperty =
            "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(";
        String requireAuth = "'derby.connection.requireAuthentication'";
        String defaultConnMode =
            "'derby.database.defaultConnectionMode'";
        String fullAccessUsers = "'derby.database.fullAccessUsers'";
        String readOnlyAccessUsers =
            "'derby.database.readOnlyAccessUsers'";
        String provider = "'derby.authentication.provider'";
        String propertiesOnly = "'derby.database.propertiesOnly'";

        System.out.println("Turning on authentication.");
        Statement s = conn.createStatement();

        // Set and confirm requireAuthentication
        s.executeUpdate(setProperty + requireAuth + ", 'true')");
        ResultSet rs = s.executeQuery(getProperty + requireAuth + ")");
        rs.next();
        System.out.println("Value of requireAuthentication is " +
            rs.getString(1));

        // Set authentication scheme to Derby builtin
        s.executeUpdate(setProperty + provider + ", 'BUILTIN')");

        // Create some sample users
        s.executeUpdate(
            setProperty + "'derby.user.sa', 'ajaxj3x9')");
        s.executeUpdate(
            setProperty + "'derby.user.guest', 'java5w6x')");
        s.executeUpdate(
            setProperty + "'derby.user.mary', 'little7xylamb')");

        // Define noAccess as default connection mode
        s.executeUpdate(
            setProperty + defaultConnMode + ", 'noAccess')");

        // Confirm default connection mode
        rs = s.executeQuery(getProperty + defaultConnMode + ")");
        rs.next();
        System.out.println("Value of defaultConnectionMode is " +
            rs.getString(1));

        // Define read-write user
        s.executeUpdate(
            setProperty + fullAccessUsers + ", 'sa,mary')");

        // Define read-only user
        s.executeUpdate(
            setProperty + readOnlyAccessUsers + ", 'guest')");

        // Confirm full-access users
        rs = s.executeQuery(getProperty + fullAccessUsers + ")");
        rs.next();
```

```
        System.out.println(
            "Value of fullAccessUsers is " + rs.getString(1));

        // Confirm read-only users
        rs = s.executeQuery(getProperty + readOnlyAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of readOnlyAccessUsers is " + rs.getString(1));

        // We would set the following property to TRUE only when we were
        // ready to deploy. Setting it to FALSE means that we can always
        // override using system properties if we accidentally paint
        // ourselves into a corner.
        s.executeUpdate(setProperty + propertiesOnly + ", 'false')");
        s.close();
    }

    /**
     * Report exceptions, with special handling of SQLExceptions,
     * and exit.
     *
     * @param e an exception (Throwable)
     */
    static void errorPrintAndExit(Throwable e) {
        if (e instanceof SQLException)
            SQLExceptionPrint((SQLException)e);
        else {
            System.out.println("A non-SQL error occurred.");
            e.printStackTrace();
        }
        System.exit(1);
    }

    /**
     * Iterate through a stack of SQLExceptions.
     *
     * @param sqle a SQLException
     */
    static void SQLExceptionPrint(SQLException sqle) {
        while (sqle != null) {
            System.out.println("\n---SQLException Caught---\n");
            System.out.println("SQLState:   " + (sqle).getSQLState());
            System.out.println("Severity: " + (sqle).getErrorCode());
            System.out.println("Message:  " + (sqle).getMessage());
            sqle = sqle.getNextException();
        }
    }
}
```

User authentication examples using SQL authorization

These examples show how to use SQL authorization in conjunction with Derby's built-in user authentication and user authorization.

These examples are based on the examples in User authentication and authorization client example and User authentication and authorization embedded example.

User authentication and SQL authorization client example

This example consists of a pair of programs, `AuthExampleClientSQLAuth1.java` and `AuthExampleClientSQLAuth2.java`, which show how to use SQL authorization, in addition to Derby's built-in user authentication and user authorization, with the client driver.

This example uses one program to set properties and a second program to perform database operations. A similar example that uses the embedded driver, in User

authentication and SQL authorization embedded example, is a single long program. Either example would work equally well in the other format.

See User authorizations for more information on using SQL authorization, which allows you to use ANSI SQL Standard GRANT and REVOKE statements. Compare this example to the one in User authentication and authorization client example, which does not use SQL authorization.

The first program, `AuthExampleClientSQLAuth1.java`, does the following:

1. Creates a database named `sqlAuthClientDB`, using the client driver. The connection URL creates the database as the user `mary`, who is therefore the database owner. After SQL authorization is enabled, only the database owner will have the right to set and read database properties.
2. Sets database properties that create users with different levels of access (no access, read-only access, and full access), that require authentication, and that turn on SQL authorization. The users `mary` and `sqlsam` have full access.
3. Closes the connection, then shuts down the database so that the authentication and SQL authorization changes can take effect.

The second program, `AuthExampleClientSQLAuth2.java`, does the following:

1. Tries to connect to the database without a username and password, raising an exception.
2. Tries to connect to the database as a user with no access, raising an exception.
3. Connects to the database as a user with read-only access; the connection succeeds, but an attempt to create a table raises an exception.
4. Connects to the database as `mary`, who has full access; this user creates and populates a table. This user also grants select and insert privileges on this table to another user.
5. Connects to the database as `sqlsam`, the user who has been granted select and insert privileges by `mary`. This user has full (that is, read-write) access on the connection level, but has limited powers for this table because SQL authorization is active. The user successfully performs select and insert operations on the table, but an attempt to delete a row from the table raises an exception.
6. Connects to the database again as `mary`, who then deletes the table.
7. Shuts down the database.

Make sure that the `javac` command is in your path, then compile the programs as follows:

```
javac AuthExampleClientSQLAuth1.java
```

```
javac AuthExampleClientSQLAuth2.java
```

Before you run the programs, start the Derby Network Server as described in step 2 of "Activity 4: Create and run a JDBC program using the client driver and Network Server" in *Getting Started with Java DB*. When you run the programs, make sure that `%DERBY_HOME%\lib\derbyclient.jar` (or `$DERBY_HOME/lib/derbyclient.jar`) is in your classpath. For example, you might use the following commands on a UNIX system:

```
java -cp .:${DERBY_HOME}/lib/derbyclient.jar
AuthExampleClientSQLAuth1
```

```
java -cp .:${DERBY_HOME}/lib/derbyclient.jar
AuthExampleClientSQLAuth2
```

Source code for `AuthExampleClientSQLAuth1.java`

```
import java.sql.*;
```

```
public class AuthExampleClientSQLAuth1 {

    public static void main(String[] args) {

        String driver = "org.apache.derby.jdbc.ClientDriver";
        String dbName="sqlAuthClientDB";
        String dbOwner="mary";
        String connectionURL = "jdbc:derby://localhost:1527/" + dbName +
            ";user=" + dbOwner + ";create=true";
        Connection conn = null;

        // Load the driver. This code is not needed if you are using
        // JDK 6, because in that environment the driver is loaded
        // automatically when the application requests a connection.
        try {
            Class.forName(driver);
            System.out.println(driver + " loaded.");
        } catch (java.lang.ClassNotFoundException ce) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(ce.getMessage());
            System.out.println("\n Make sure your CLASSPATH variable " +
                "contains %DERBY_HOME%\\lib\\derbyclient.jar " +
                "(${DERBY_HOME}/lib/derbyclient.jar).\n");
        } catch (Exception ee) {
            errorPrintAndExit(ee);
        }

        // Create and boot the database as user mary (who then becomes
        // the database owner), set up users and then shut down the
        // database
        try {
            System.out.println("Trying to connect to " + connectionURL);
            conn = DriverManager.getConnection(connectionURL);
            System.out.println("Connected to database " + connectionURL);

            turnOnBuiltInUsers(conn);

            // Close connection
            conn.close();
            System.out.println("Closed connection");

            /* Shut down the database to make static properties take
             * effect. Because the default connection mode is now
             * noAccess, you must specify a user that has access. But
             * because requireAuthentication and sqlAuthorization do not
             * take effect until you restart the database, you do not
             * need to specify a password.
             *
             * Database shutdown throws the 08006 exception to confirm
             * success.
             */
            try {
                DriverManager.getConnection(
                    "jdbc:derby://localhost:1527/" + dbName +
                    ";user=mary;shutdown=true");
            } catch (SQLException se)  {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");
        } catch (Throwable e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Turn on built-in user authentication and SQL authorization.
     *
     * Default connection mode is fullAccess, but SQL authorization
```

```
     * restricts access to the owners of database objects.
     *
     * @param conn a connection to the database
     */
    public static void turnOnBuiltInUsers(Connection conn)
            throws SQLException {

        String setProperty =
            "CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(";
        String getProperty =
            "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(";
        String requireAuth = "'derby.connection.requireAuthentication'";
        String sqlAuthorization = "'derby.database.sqlAuthorization'";
        String defaultConnMode =
            "'derby.database.defaultConnectionMode'";
        String fullAccessUsers = "'derby.database.fullAccessUsers'";
        String readOnlyAccessUsers =
            "'derby.database.readOnlyAccessUsers'";
        String provider = "'derby.authentication.provider'";
        String propertiesOnly = "'derby.database.propertiesOnly'";

        System.out.println(
            "Turning on authentication and SQL authorization.");
        Statement s = conn.createStatement();

        // Set requireAuthentication
        s.executeUpdate(setProperty + requireAuth + ", 'true')");
        // Set sqlAuthorization
        s.executeUpdate(setProperty + sqlAuthorization + ", 'true')");

        // Retrieve and display property values
        ResultSet rs = s.executeQuery(getProperty + requireAuth + ")");
        rs.next();
        System.out.println(
            "Value of requireAuthentication is " + rs.getString(1));

        rs = s.executeQuery(getProperty + sqlAuthorization + ")");
        rs.next();
        System.out.println(
            "Value of sqlAuthorization is " + rs.getString(1));

        // Set authentication scheme to Derby builtin
        s.executeUpdate(setProperty + provider + ", 'BUILTIN')");

        // Create some sample users
        s.executeUpdate(
            setProperty + "'derby.user.sa', 'ajaxj3x9')");
        s.executeUpdate(
            setProperty + "'derby.user.guest', 'java5w6x')");
        s.executeUpdate(
            setProperty + "'derby.user.mary', 'little7xylamb')");
        s.executeUpdate(
            setProperty + "'derby.user.sqlsam', 'light8q9bulb')");

        // Define noAccess as default connection mode
        s.executeUpdate(
            setProperty + defaultConnMode + ", 'noAccess')");

        // Confirm default connection mode
        rs = s.executeQuery(getProperty + defaultConnMode + ")");
        rs.next();
        System.out.println("Value of defaultConnectionMode is " +
            rs.getString(1));

        // Define read-write users
        s.executeUpdate(
            setProperty + fullAccessUsers + ", 'sqlsam,mary')");

        // Define read-only user
        s.executeUpdate(
```

```
            setProperty + readOnlyAccessUsers + ", 'guest')");

        // Therefore, user sa has no access

        // Confirm full-access users
        rs = s.executeQuery(getProperty + fullAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of fullAccessUsers is " + rs.getString(1));

        // Confirm read-only users
        rs = s.executeQuery(getProperty + readOnlyAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of readOnlyAccessUsers is " + rs.getString(1));

        // We would set the following property to TRUE only when we were
        // ready to deploy. Setting it to FALSE means that we can always
        // override using system properties if we accidentally paint
        // ourselves into a corner.
        s.executeUpdate("CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(" +
            "'derby.database.propertiesOnly', 'false')");
        s.close();
    }

    /**
     * Report exceptions, with special handling of SQLExceptions,
     * and exit.
     *
     * @param e an exception (Throwable)
     */
    static void errorPrintAndExit(Throwable e) {
        if (e instanceof SQLException)
            SQLExceptionPrint((SQLException)e);
        else {
            System.out.println("A non-SQL error occurred.");
            e.printStackTrace();
        }
        System.exit(1);
    }

    /**
     * Iterate through a stack of SQLExceptions.
     *
     * @param sqle a SQLException
     */
    static void SQLExceptionPrint(SQLException sqle) {
        while (sqle != null) {
            System.out.println("\n---SQLException Caught---\n");
            System.out.println("SQLState:   " + (sqle).getSQLState());
            System.out.println("Severity: " + (sqle).getErrorCode());
            System.out.println("Message:  " + (sqle).getMessage());
            sqle = sqle.getNextException();
        }
    }
}
```

Source code for `AuthExampleClientSQLAuth2.java`

```
import java.sql.*;

public class AuthExampleClientSQLAuth2 {

    public static void main(String[] args) {

        String driver = "org.apache.derby.jdbc.ClientDriver";
        String dbName="sqlAuthClientDB";
        String dbOwner="mary";
        String connectionURL = "jdbc:derby://localhost:1527/" + dbName;
        Connection conn = null;
```

```
        // Restart database and confirm that unauthorized users cannot
        //  access it

        // Load the driver. This code is not needed if you are using
        // JDK 6, because in that environment the driver is loaded
        // automatically when the application requests a connection.
        try {
            Class.forName(driver);
            System.out.println(driver + " loaded.");
        } catch (java.lang.ClassNotFoundException ce) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(ce.getMessage());
            System.out.println("\n Make sure your CLASSPATH variable " +
                "contains %DERBY_HOME%\\lib\\derbyclient.jar " +
                "(${DERBY_HOME}/lib/derbyclient.jar). \n");
        } catch (Exception ee) {
            errorPrintAndExit(ee);
        }

        // Try to log in with no username or password
        try {
            // connection attempt should fail
            System.out.println("Trying to connect to " + connectionURL +
                " without username or password");
            conn = DriverManager.getConnection(connectionURL);
            System.out.println(
                "ERROR: Unexpectedly connected to database " + dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("08004")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
            } else {
                errorPrintAndExit(e);
            }
        }

        // Try to log in as a valid user with noAccess
        try {
            // connection attempt should fail
            String newURL = connectionURL + ";user=sa;password=ajaxj3x9";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println(
                "ERROR: Unexpectedly allowed to connect to database " +
                     dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("08004")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with read-only access
        try {
            // connection should succeed, but create table should fail
            String newURL = connectionURL +
                ";user=guest;password=java5w6x";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName +
                " with read-only access");

            Statement s = conn.createStatement();
            s.executeUpdate(
                "CREATE TABLE accessibletbl(textcol VARCHAR(6))");
```

```
            System.out.println(
                "ERROR: Unexpectedly allowed to modify database " +
                     dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("25503")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
                try {
                    conn.close();
                } catch (SQLException ee) {
                    errorPrintAndExit(ee);
                }
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with full access
        // Create, update, and query table
        // Grant select and insert privileges to another user
        try {
            // this should succeed
            String newURL = connectionURL +
                ";user=mary;password=little7xylamb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            Statement s = conn.createStatement();

            s.executeUpdate(
                "CREATE TABLE accessibletbl(textcol VARCHAR(6))");
            System.out.println("Created table accessibletbl");
            s.executeUpdate("INSERT INTO accessibletbl VALUES('hello')");

            ResultSet rs = s.executeQuery("SELECT * FROM accessibletbl");
            rs.next();
            System.out.println("Value of accessibletbl/textcol is " +
                rs.getString(1));

            // grant insert privileges on table to user sqlsam
            s.executeUpdate(
                "GRANT SELECT, INSERT ON accessibletbl TO sqlsam");
            System.out.println(
                "Granted select/insert privileges to sqlsam");

            s.close();
            conn.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        // Log in as user with select and insert privileges on the table,
        //  but not delete privileges
        try {
            String newURL =
                connectionURL + ";user=sqlsam;password=light8q9bulb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            // look at table
            Statement s = conn.createStatement();
            ResultSet rs =
                s.executeQuery("SELECT * FROM mary.accessibletbl");
            rs.next();
            System.out.println("Value of accessibletbl/textcol is " +
                rs.getString(1));
```

```
            s.executeUpdate(
                "INSERT INTO mary.accessibletbl VALUES('sam')");
            System.out.println("Inserted string into table");

            rs = s.executeQuery("SELECT * FROM mary.accessibletbl");
            while (rs.next()) {
               System.out.println("Value of accessibletbl/textcol is " +
                   rs.getString(1));
            }
            s.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        try {
            Statement s = conn.createStatement();

            // this should fail
            s.executeUpdate("DELETE FROM mary.accessibletbl " +
                "WHERE textcol = 'hello'");
            System.out.println("ERROR: Unexpectedly allowed to DELETE " +
                "table mary.accessibletbl");
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("42500")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
                try {
                    conn.close();
                } catch (SQLException ee) {
                    errorPrintAndExit(ee);
                }
            } else {
                errorPrintAndExit(e);
            }
        }

        /* Log in again as mary, delete table
         */
        try {
            String newURL = connectionURL +
                ";user=mary;password=little7xylamb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            Statement s = conn.createStatement();
            s.executeUpdate("DROP TABLE accessibletbl");
            System.out.println("Removed table accessibletbl");
            s.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        try {
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /** Close connection and shut down database.
     *
     * @param conn a connection to the database
     */
    public static void cleanUpAndShutDown (Connection conn)
            throws SQLException {

        String dbName="sqlAuthClientDB";
        String dbOwner="mary";
```

```
        String connectionURL = "jdbc:derby://localhost:1527/" + dbName;

        try {
            conn.close();
            System.out.println("Closed connection");

            // As mary, the database owner, shut down the database.
            try {
                String newURL = connectionURL + ";user=" + dbOwner +
                    ";password=little7xylamb;shutdown=true";
                DriverManager.getConnection(newURL);
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Report exceptions, with special handling of SQLExceptions,
     * and exit.
     *
     * @param e an exception (Throwable)
     */
    static void errorPrintAndExit(Throwable e) {
        if (e instanceof SQLException)
            SQLExceptionPrint((SQLException)e);
        else {
            System.out.println("A non-SQL error occurred.");
            e.printStackTrace();
        }
        System.exit(1);
    }

    /**
     * Iterate through a stack of SQLExceptions.
     *
     * @param sqle a SQLException
     */
    static void SQLExceptionPrint(SQLException sqle) {
        while (sqle != null) {
            System.out.println("\n---SQLException Caught---\n");
            System.out.println("SQLState:   " + (sqle).getSQLState());
            System.out.println("Severity: " + (sqle).getErrorCode());
            System.out.println("Message:  " + (sqle).getMessage());
            sqle = sqle.getNextException();
        }
    }
}
```

User authentication and SQL authorization embedded example

This example, `AuthExampleEmbeddedSQLAuth.java`, shows how to use SQL authorization, in addition to Derby's built-in user authentication and user authorization, with the embedded driver.

This example is a single long program. A similar example that uses the client driver, in User authentication and SQL authorization client example, uses one program to set properties and a second program to perform database operations. Either example would work equally well in the other format.

See User authorizations for more information on using SQL authorization, which allows you to use ANSI SQL Standard GRANT and REVOKE statements. Compare this

example to the one in User authentication and authorization embedded example, which does not use SQL authorization.

The program does the following:

1. Starts Derby and creates a database named `sqlAuthEmbDB`, using the embedded driver. The connection URL creates the database as the user `mary`, who is therefore the database owner. After SQL authorization is enabled, only the database owner will have the right to set and read database properties.
2. Sets database properties that create users with different levels of access (no access, read-only access, and full access), that require authentication, and that turn on SQL authorization. The users `mary` and `sqlsam` have full access.
3. Closes the connection, then stops and restarts the database so that the authentication and SQL authorization changes can take effect.
4. Tries to connect to the database without a username and password, raising an exception.
5. Tries to connect to the database as a user with no access, raising an exception.
6. Connects to the database as a user with read-only access; the connection succeeds, but an attempt to create a table raises an exception.
7. Connects to the database as `mary`, who has full access; this user creates and populates a table. This user also grants select and insert privileges on this table to another user.
8. Connects to the database as `sqlsam`, the user who has been granted select and insert privileges by `mary`. This user has full (that is, read-write) access on the connection level, but has limited powers for this table because SQL authorization is active. The user successfully performs select and insert operations on the table, but an attempt to delete a row from the table raises an exception.
9. Connects to the database again as `mary`, who then deletes the table.
10. Closes the connection, shuts down the database, then shuts down Derby.

Make sure that the `javac` command is in your path, then compile the program as follows:

```
javac AuthExampleEmbeddedSQLAuth.java
```

When you run `AuthExampleEmbeddedSQLAuth`, make sure that `%DERBY_HOME%\lib\derby.jar` (or `$DERBY_HOME/lib/derby.jar`) is in your classpath. For example, you might use the following command on a Windows system:

```
java -cp .;%DERBY_HOME%\lib\derby.jar AuthExampleEmbeddedSQLAuth
```

Source code for `AuthExampleEmbeddedSQLAuth.java`

```
import java.sql.*;

public class AuthExampleEmbeddedSQLAuth {

    public static void main(String[] args) {

        String driver = "org.apache.derby.jdbc.EmbeddedDriver";
        String dbName="sqlAuthEmbDB";
        String dbOwner="mary";
        String connectionURL = "jdbc:derby:" + dbName +
            ";user=" + dbOwner + ";create=true";
        Connection conn = null;

        // Load the driver. This code is not needed if you are using
        // JDK 6, because in that environment the driver is loaded
        // automatically when the application requests a connection.
        try {
            Class.forName(driver);
            System.out.println(driver + " loaded.");
        } catch (java.lang.ClassNotFoundException ce) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(ce.getMessage());
```

```
            System.out.println("\n Make sure your CLASSPATH variable " +
                "contains %DERBY_HOME%\\lib\\derby.jar " +
                "(${DERBY_HOME}/lib/derby.jar).\n");
        } catch (Exception ee) {
            errorPrintAndExit(ee);
        }

        // Create and boot the database as user mary (who then becomes
        // the database owner), set up users and then shut down the
        // database
        try {
            System.out.println("Trying to connect to " + connectionURL);
            conn = DriverManager.getConnection(connectionURL);
            System.out.println("Connected to database " + connectionURL);

            turnOnBuiltInUsers(conn);

            // Close connection
            conn.close();
            System.out.println("Closed connection");

            /* Shut down the database to make static properties take
             * effect. Because the default connection mode is now
             * noAccess, you must specify a user that has access. But
             * because requireAuthentication and sqlAuthorization do not
             * take effect until you restart the database, you do not
             * need to specify a password.
             *
             * Database shutdown throws the 08006 exception to confirm
             * success.
             */
            try {
                DriverManager.getConnection("jdbc:derby:" + dbName +
                    ";user=mary;shutdown=true");
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        // Restart database and confirm that unauthorized users cannot
        //  access it
        connectionURL = "jdbc:derby:" + dbName;

        // Try to log in with no username or password
        try {
            // connection attempt should fail
            System.out.println("Trying to connect to " + connectionURL +
                " without username or password");
            conn = DriverManager.getConnection(connectionURL);
            System.out.println(
                "ERROR: Unexpectedly connected to database " + dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("08004")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
            } else {
                errorPrintAndExit(e);
            }
        }

        // Try to log in as a valid user with noAccess
        try {
            // connection attempt should fail
            String newURL = connectionURL + ";user=sa;password=ajaxj3x9";
```

```
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println(
                "ERROR: Unexpectedly allowed to connect to database " +
                     dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("08004")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with read-only access
        try {
            // connection should succeed, but create table should fail
            String newURL = connectionURL +
                ";user=guest;password=java5w6x";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName +
                " with read-only access");

            Statement s = conn.createStatement();
            s.executeUpdate(
                "CREATE TABLE accessibletbl(textcol VARCHAR(6))");
            System.out.println(
                "ERROR: Unexpectedly allowed to modify database " +
                     dbName);
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("25503")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
                try {
                    conn.close();
                } catch (SQLException ee) {
                    errorPrintAndExit(ee);
                }
            } else {
                errorPrintAndExit(e);
            }
        }

        // Log in as a user with full access
        // Create, update, and query table
        // Grant select and insert privileges to another user
        try {
            // this should succeed
            String newURL = connectionURL +
                ";user=mary;password=little7xylamb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            Statement s = conn.createStatement();

            s.executeUpdate(
                "CREATE TABLE accessibletbl(textcol VARCHAR(6))");
            System.out.println("Created table accessibletbl");
            s.executeUpdate("INSERT INTO accessibletbl VALUES('hello')");

            ResultSet rs = s.executeQuery("SELECT * FROM accessibletbl");
            rs.next();
            System.out.println("Value of accessibletbl/textcol is " +
                rs.getString(1));

            // grant insert privileges on table to user sqlsam
```

```
            s.executeUpdate(
                "GRANT SELECT, INSERT ON accessibletbl TO sqlsam");
            System.out.println(
                "Granted select/insert privileges to sqlsam");

            s.close();
            conn.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        // Log in as user with select and insert privileges on the table,
        //  but not delete privileges
        try {
            String newURL =
                connectionURL + ";user=sqlsam;password=light8q9bulb";
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            // look at table
            Statement s = conn.createStatement();
            ResultSet rs =
                s.executeQuery("SELECT * FROM mary.accessibletbl");
            rs.next();
            System.out.println("Value of accessibletbl/textcol is " +
                rs.getString(1));

            s.executeUpdate(
                "INSERT INTO mary.accessibletbl VALUES('sam')");
            System.out.println("Inserted string into table");

            rs = s.executeQuery("SELECT * FROM mary.accessibletbl");
            while (rs.next()) {
               System.out.println("Value of accessibletbl/textcol is " +
                   rs.getString(1));
            }
            s.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        try {
            Statement s = conn.createStatement();

            // this should fail
            s.executeUpdate("DELETE FROM mary.accessibletbl " +
                "WHERE textcol = 'hello'");
            System.out.println("ERROR: Unexpectedly allowed to DELETE " +
                "table mary.accessibletbl");
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            if (e.getSQLState().equals("42500")) {
                System.out.println("Correct behavior: SQLException: " +
                    e.getMessage());
                try {
                    conn.close();
                } catch (SQLException ee) {
                    errorPrintAndExit(ee);
                }
            } else {
                errorPrintAndExit(e);
            }
        }

        /* Log in again as mary, delete table
         */
        try {
            String newURL = connectionURL +
                ";user=mary;password=little7xylamb";
```

```
            System.out.println("Trying to connect to " + newURL);
            conn = DriverManager.getConnection(newURL);
            System.out.println("Connected to database " + dbName);

            Statement s = conn.createStatement();
            s.executeUpdate("DROP TABLE accessibletbl");
            System.out.println("Removed table accessibletbl");
            s.close();
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }

        try {
            cleanUpAndShutDown(conn);
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Close connection and shut down database. Since this is embedded
     * mode, we must also shut down the Derby system.
     *
     * @param conn a connection to the database
     */
    public static void cleanUpAndShutDown (Connection conn)
            throws SQLException {

        String dbName="sqlAuthEmbDB";
        String dbOwner="mary";
        String connectionURL = "jdbc:derby:" + dbName;

        try {
            conn.close();
            System.out.println("Closed connection");

            // As mary, the database owner, shut down the database.
            try {
                String newURL = connectionURL + ";user=" + dbOwner +
                    ";password=little7xylamb;shutdown=true";
                DriverManager.getConnection(newURL);
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("08006") ) {
                    throw se;
                }
            }
            System.out.println("Database shut down normally");

            try {
                DriverManager.getConnection("jdbc:derby:;shutdown=true");
            } catch (SQLException se) {
                if ( !se.getSQLState().equals("XJ015") ) {
                    throw se;
                }
            }

            System.out.println("Derby system shut down normally");
        } catch (SQLException e) {
            errorPrintAndExit(e);
        }
    }

    /**
     * Turn on built-in user authentication and SQL authorization.
     *
     * Default connection mode is fullAccess, but SQL authorization
     * restricts access to the owners of database objects.
     *
     * @param conn a connection to the database
     */
```

```
    public static void turnOnBuiltInUsers(Connection conn)
            throws SQLException {

        String setProperty =
            "CALL SYSCS_UTIL.SYSCS_SET_DATABASE_PROPERTY(";
        String getProperty =
            "VALUES SYSCS_UTIL.SYSCS_GET_DATABASE_PROPERTY(";
        String requireAuth = "'derby.connection.requireAuthentication'";
        String sqlAuthorization = "'derby.database.sqlAuthorization'";
        String defaultConnMode =
            "'derby.database.defaultConnectionMode'";
        String fullAccessUsers = "'derby.database.fullAccessUsers'";
        String readOnlyAccessUsers =
            "'derby.database.readOnlyAccessUsers'";
        String provider = "'derby.authentication.provider'";
        String propertiesOnly = "'derby.database.propertiesOnly'";

        System.out.println(
            "Turning on authentication and SQL authorization.");
        Statement s = conn.createStatement();

        // Set requireAuthentication
        s.executeUpdate(setProperty + requireAuth + ", 'true')");
        // Set sqlAuthorization
        s.executeUpdate(setProperty + sqlAuthorization + ", 'true')");

        // Retrieve and display property values
        ResultSet rs = s.executeQuery(getProperty + requireAuth + ")");
        rs.next();
        System.out.println(
            "Value of requireAuthentication is " + rs.getString(1));

        rs = s.executeQuery(getProperty + sqlAuthorization + ")");
        rs.next();
        System.out.println(
            "Value of sqlAuthorization is " + rs.getString(1));

        // Set authentication scheme to Derby builtin
        s.executeUpdate(setProperty + provider + ", 'BUILTIN')");

        // Create some sample users
        s.executeUpdate(
            setProperty + "'derby.user.sa', 'ajaxj3x9')");
        s.executeUpdate(
            setProperty + "'derby.user.guest', 'java5w6x')");
        s.executeUpdate(
            setProperty + "'derby.user.mary', 'little7xylamb')");
        s.executeUpdate(
            setProperty + "'derby.user.sqlsam', 'light8q9bulb')");

        // Define noAccess as default connection mode
        s.executeUpdate(
            setProperty + defaultConnMode + ", 'noAccess')");

        // Confirm default connection mode
        rs = s.executeQuery(getProperty + defaultConnMode + ")");
        rs.next();
        System.out.println("Value of defaultConnectionMode is " +
            rs.getString(1));

        // Define read-write users
        s.executeUpdate(
            setProperty + fullAccessUsers + ", 'sqlsam,mary')");

        // Define read-only user
        s.executeUpdate(
            setProperty + readOnlyAccessUsers + ", 'guest')");

        // Therefore, user sa has no access
```

```
        // Confirm full-access users
        rs = s.executeQuery(getProperty + fullAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of fullAccessUsers is " + rs.getString(1));

        // Confirm read-only users
        rs = s.executeQuery(getProperty + readOnlyAccessUsers + ")");
        rs.next();
        System.out.println(
            "Value of readOnlyAccessUsers is " + rs.getString(1));

        // We would set the following property to TRUE only when we were
        // ready to deploy. Setting it to FALSE means that we can always
        // override using system properties if we accidentally paint
        // ourselves into a corner.
        s.executeUpdate(setProperty + propertiesOnly + ", 'false')");
        s.close();
    }

    /**
     * Report exceptions, with special handling of SQLExceptions,
     * and exit.
     *
     * @param e an exception (Throwable)
     */
    static void errorPrintAndExit(Throwable e) {
        if (e instanceof SQLException)
            SQLExceptionPrint((SQLException)e);
        else {
            System.out.println("A non-SQL error occurred.");
            e.printStackTrace();
        }
        System.exit(1);
    }

    /**
     * Iterate through a stack of SQLExceptions.
     *
     * @param sqle a SQLException
     */
    static void SQLExceptionPrint(SQLException sqle) {
        while (sqle != null) {
            System.out.println("\n---SQLException Caught---\n");
            System.out.println("SQLState:   " + (sqle).getSQLState());
            System.out.println("Severity: " + (sqle).getErrorCode());
            System.out.println("Message:  " + (sqle).getMessage());
            sqle = sqle.getNextException();
        }
    }
}
```

Running Derby under a security manager

When running within an application or application server with a Java 2 Security Manager enabled, Derby must be granted certain permissions to execute and access database files.

For more information about permissions and examples of creating permission objects and granting permissions, see the Security Architecture specification at *http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html*.

Granting permissions to Derby

This section discusses which permissions should be granted to Derby (the code base `derby.jar`).

See Default Policy Implementation and Policy File Syntax at http://java.sun.com/j2se/1.4.2/docs/guide/security/PolicyFiles.html for more information about creating policy files.

Mandatory permissions

permission java.lang.RuntimePermission createClassLoader
Mandatory. It allows Derby to execute SQL queries and supports loading class files from jar files stored in the database.

permission java.util.PropertyPermission "derby.*", "read"
Allows Derby to read individual Derby properties set in the JVM machine's system set. If the action is denied, properties in the JVM machine's system set are ignored.

permission java.util.PropertyPermission "derby.storage.jvmInstanceId", "write"
This property is used by Derby to prevent the accidental boot of the database by two class loaders. If the database is booted by two class loaders, database corruption can occur. If write permission for this property is not granted, a message is printed to the log file which indicates that the Derby database is not protected from dual boot and possible corruption.

Database access permissions

permission java.io.FilePermission "directory${/}/-", "read,write,delete"
Allows Derby to manage files within the database that maps to the directory specified. For read-only databases, only the "read" action needs to be granted.

Optional permissions

permission java.io.FilePermission "${derby.system.home}", "read,write"
Allows Derby to determine the system directory when set by *derby.system.home* and create it if needed. If the system directory already exists then only the "read" permission needs to be granted.

permission java.util.PropertyPermission "user.dir", "read"
Permits access to the system directory value if *derby.system.home* is not set or no permission has been granted to read the *derby.system.home* property.

permission java.util.PropertyPermission "sun.arch.data.model", "read"
If set by the JVM, this is the definite answer to whether the system is 32 or 64 bit.

permission java.util.PropertyPermission "os.arch", "read"
Used by Derby to determine if the system is 32 or 64 bit, if the system property *sun.arch.data.model* isn't set by the JVM. Derby has to recognize the value of *os.arch* to determine if the system is 32 or 64 bit, and if the value isn't recognized a heuristic will be used instead.

permission java.io.FilePermission "${derby.system.home}${/}derby.properties", "read"
Allows Derby to read the system properties file from the system directory.

permission java.io.FilePermission "${derby.system.home}${/}derby.log", "read,write,delete"

permission java.io.FilePermission "${user.dir}${/}derby.log", "read,write,delete";
Only one of these permissions is needed. Permits the application to read, write, and delete to the Derby log file, unless the log has been re-directed. (See the *derby.stream.error* properties in the *Java DB Reference Manual* for more information.) If one of the requested valid actions is denied, the Derby log will be `java.lang.System.err`.

permission java.security.SecurityPermission "getPolicy"
You need this permission if you want to change the security policy on the fly and reload it into a running system. Given this permission, a DBA can reload the policy file by calling the `SYSCS_UTIL.SYSCS_RELOAD_SECURITY_POLICY` system procedure. For more information, see the section which describes this procedure in the *Java DB Reference Manual*.

permission java.lang.RuntimePermission "setContextClassLoader"
Allows Derby to set the context class loader for long running threads to null to avoid potential for class loader leaks in application server environments when the application server starts Derby in a custom class loader.

permission java.lang.RuntimePermission "getClassLoader"
This permission is also needed when setting the context class loader to avoid class loader leaks. The class loader for the parent is saved and set to null before creation of the thread and restored afterwards.

permission javax.management.MBeanServerPermission "createMBeanServer";
Allows Derby to create an MBean server. If the JVM running Derby supports the platform MBean server, Derby will automatically try to create such a server if it does not already exist. For details, visit the wiki page http://wiki.apache.org/db-derby/DerbyJMX.

permission javax.management.MBeanPermission "org.apache.derby.*#[org.apache.derby:*]","registerMBean,unregisterMBean";
Allows Derby to register and unregister its (JMX) MBeans. Such MBeans are associated with the domain `org.apache.derby`, which is also the prefix of the fully qualified class name of all Derby MBeans. For more information about Derby's MBeans, refer to the public API (Javadoc) documentation of the package `org.apache.derby.mbeans` and its subpackages. It is possible to fine-tune this permission, for example in order to allow access only to certain MBeans. To fine-tune this permission, see the API documentation for `javax.management.MBeanPermission` or the JMX Instrumentation and Agent Specification.

permission javax.management.MBeanTrustPermission "register";
Trusts Derby code to be the source of MBeans and to register these in the MBean server.

Combining permissions

You might grant one *FilePermission* that encompasses several or all of the permissions instead of separately granting a number of the more specific permissions. For example:

```
permission java.io.FilePermission "${derby.system.home}/-",
 "read,write,delete";
```

This allows the Derby engine complete access to the system directory and any databases contained in the system directory.

Examples of Java 2 security policy files for embedded Derby

Java 2 security policy file example 1

```
/* Grants permission to run Derby and access all      */
/* databases under the Derby system home              */
/* when it is specified by the system property             */
/* Derby.system.home                                        */

/* Note Derby.system.home must be an absolute pathname */

grant codeBase "file://f:/derby/lib/derby.jar" {

  permission java.lang.RuntimePermission "createClassLoader";
  permission java.util.PropertyPermission "derby.*", "read";
  permission.java.io.FilePermission "${derby.system.home}","read";
  permission java.io.FilePermission "${derby.system.home}${/}
     -", "read,write,delete";
  permission java.util.PropertyPermission "derby.storage.jvmInstanceId",
     "write";
};
```

Java 2 security policy file example 2

```
/* Grants permission to run Derby and access all      */
/* databases under the Derby system home              */
/* when it defaults to the current directory               */


grant codeBase "file://f:/derby/lib/derby.jar" {
  permission java.lang.RuntimePermission "createClassLoader";
  permission java.util.PropertyPermission "derby.*", "read";
  permission java.util.PropertyPermission "user.dir", "read";
  permission java.io.FilePermission "${derby.system.home}","read";
  permission java.io.FilePermission "${user.dir}${/}-",
      "read,write,delete";
  permission java.util.PropertyPermission "derby.storage.jvmInstanceId",
      "write";

};
```

Java 2 security policy file example 3

```
/* Grants permission to run Derby and access a single */
/* database (salesdb) under the Derby system home     */

/* Note Derby.system.home must be an absolute pathname */

grant codeBase "file://f:/derby/lib/derby.jar" {

  permission java.lang.RuntimePermission "createClassLoader";
  permission java.util.PropertyPermission "derby.*", "read";
  permission java.io.FilePermission "${derby.system.home}","read";
  permission java.io.FilePermission "${derby.system.home}${/}*",
      "read,write,delete";
  permission java.io.FilePermission "${derby.system.home}${/}
      salesdb${/}-", "read,write,delete";
  permission java.util.PropertyPermission "derby.storage.jvmInstanceId",
      "write";


};
```

Developing tools and using Derby with an IDE

Applications such as database tools are designed to work with databases whose schemas and contents are unknown in advance. This section discusses a few topics useful for such applications.

Offering connection choices to the user

JDBC's *java.sql.Driver.getPropertyInfo* method allows a generic GUI tool to determine the properties for which it should prompt a user in order to get enough information to connect to a database. Depending on the values the user has supplied so far, additional values might become necessary. It might be necessary to iterate though several calls to *getPropertyInfo*.

If no more properties are necessary, the call returns an array of zero length.

In a Derby system, do not use the method against an instance of *org.apache.derby.jdbc.EmbeddedDriver*. Instead, request the JDBC driver from the driver manager:

```
java.sql.DriverManager.getDriver(
    "jdbc:derby:").getPropertyInfo(URL, Prop)
```

In a Derby system, the properties returned in the *DriverPropertyInfo* object are connection URL attributes, including a list of booted databases in a system (the *databaseName* attribute).

Databases in a system are not automatically booted until you connect with them. You can configure your system to retain the former behavior, in which case the steps described in this section will continue to work. See "*derby.system.bootAll*" in the *Java DB Reference Manual*.

getPropertyInfo requires a connection URL and a *Properties* object as parameters. Typically, what you pass are values that you will use in a future call to *java.sql.DriverManager.getConnection* when you actually connect to the database.

A call to *getPropertyInfo* with parameters that contain sufficient information to connect successfully returns an array of zero length. (Receiving this zero-length array does not *guarantee* that the *getConnection* call will succeed, because something else could go wrong.)

Repeat calls to *getPropertyInfo* until it returns a zero-length array or none of the properties remaining are desired.

The DriverPropertyInfo Array

When a non-zero-length array is returned by *getPropertyInfo*, each element is a `DriverPropertyInfo` object representing a connection URL attribute that has not already been specified. Only those that make sense in the current context are returned.

This *DriverPropertyInfo* object contains:

- *name of the attribute*
- *description*
- *current value*

  If an attribute has a default value, this is set in the value field of *DriverPropertyInfo*, even if the attribute has not been set in the connection URL or the `Properties`

object. If the attribute does not have a default value and it is not set in the URL or the `Properties` object, its value will be null.
- *list of choices*
- *whether required for a connection request*

Several fields in a *DriverPropertyInfo* object are allowed to be null.

DriverPropertyInfo array example

Here is some example code:

```
import java.sql.*;
import java.util.Properties;
// start with the least amount of information
// to see the full list of choices
// we could also enter with a URL and Properties
// provided by a user.
String url = "jdbc:derby:";
Properties info = new Properties();
Driver cDriver = DriverManager.getDriver(url);
for (;;)
        {
        DriverPropertyInfo[] attributes = cDriver.getPropertyInfo(
            url, info);
         // zero length means a connection attempt can be made
        if (attributes.length == 0)
        break;
        // insert code here to process the array, e.g.,
        // display all options in a GUI and allow the user to
        // pick and then set the attributes in info or URL.
        }
// try the connection
Connection conn = DriverManager.getConnection(url, info);
```

Using Derby with IDEs

When you use an integrated development environment (IDE) to develop an embedded Derby application, you might need to run Derby within a server framework.

This is because an IDE might try connecting to the database from two different JVMs, whereas only a single JVM instance should connect to a Derby database at one time (multiple connections from the same JVM are allowed).

An "embedded Derby application" is one which runs in the same JVM as the application. Such an application uses the embedded Derby driver (*org.apache.derby.jdbc.EmbeddedDriver*) and connection URL (*`jdbc:derby:databaseName`*). If you use this driver name or connection URL from the IDE, when the IDE tries to open a second connection to the same database with the embedded Derby, the attempt fails. Two JVMs cannot connect to the same database in embedded mode.

IDEs and multiple JVMs

When you use an integrated development environment (IDE) to build a Java application, you can launch the application from within the IDE at any point in the development process. Typically, the IDE launches a JVM dedicated to the application. When the application completes, the JVM exits. Any database connections established by the application are closed.

During the development of a database application, most IDEs allow you to test individual database connections and queries without running the entire application. When you test an individual database connection or query (which requires a database connection), the IDE might launch a JVM that runs in a specialized testing environment. In this case,

when a test completes, the JVM remains active and available for further testing, and the database connection established during the test remains open.

Because of the behaviors of the IDE described above, if you use the embedded Derby JDBC driver, you may encounter errors connecting in the following situations:

- You test an individual query or database connection and then try to run an application that uses the same database as the tested feature.

  The database connection established by testing the connection or query stays open, and prevents the application from establishing a connection to the same database.
- You run an application, and before it completes (for example, while it waits for user input), you attempt to run a second application or to test a connection or query that uses the same database as the first application.

SQL tips

This section provides some examples of interesting SQL features. It also includes a few non-SQL tips.

Retrieving the database connection URL

Derby does not have a built-in function that returns the name of the database. However, you can use `DatabaseMetaData` to return the connection URL of any local `Connection`.

```
/* in java */
String myURL = conn.getMetaData().getURL();
```

Supplying a parameter only once

If you want to supply a parameter value once and use it multiple times within a query, put it in the FROM clause with an appropriate CAST.

```
SELECT  phonebook.*
 FROM phonebook, (VALUES (CAST(? AS INT), CAST(? AS VARCHAR(255))))
      AS Choice(choice, search_string)
 WHERE search_string = (case when choice = 1 then firstnme
                 when choice=2 then lastname
      when choice=3 then phonenumber end);
```

This query selects what the second parameter will be compared to based on the value in the first parameter. Putting the parameters in the FROM clause means that they need to be applied only once to the query, and you can give them names so that you can refer to them elsewhere in the query. In the example above, the first parameter is given the name *choice*, and the second parameter is given the name *search_string*.

Defining an identity column

An identity column is a column that stores numbers that increment by one with each insertion. Identity columns are sometimes called autoincrement columns.

Derby provides autoincrement as a built-in feature; see CREATE TABLE statement in the *Java DB Reference Manual*.

Below is an example that shows how to use an identity column to create the MAP_ID column of the MAPS table in the *toursDB database*.

```
CREATE TABLE MAPS
(
MAP_ID INTEGER NOT NULL GENERATED ALWAYS AS IDENTITY (START WITH 1,
 INCREMENT BY 1),
MAP_NAME VARCHAR(24) NOT NULL,
REGION VARCHAR(26),
AREA DECIMAL(8,4) NOT NULL,
PHOTO_FORMAT VARCHAR(26) NOT NULL,
PICTURE BLOB(102400),
UNIQUE (MAP_ID, MAP_NAME)
)
```

Using third-party tools

You can hook into any JDBC tool with just our JDBC Driver class name (*org.apache.derby.jdbc.EmbeddedDriver*) and Derby's JDBC connection URL.

Tricks of the VALUES clause

This section contains some tips to use with the VALUES clause.

Multiple rows

Derby supports the complete SQL-92 VALUES clause; this is very handy in several cases.

The first useful case is that it can be used to insert multiple rows:

```
INSERT INTO OneColumnTable VALUES 1,2,3,4,5,6,7,8

INSERT INTO TwoColumnTable VALUES
    (1, 'first row'),
    (2, 'second row'),
    (3, 'third row')
```

Dynamic parameters reduce the number of times execute requests are sent across:

```
ij> -- send 5 rows at a time:
ij> PREPARE p1 AS 'INSERT INTO ThreeColumnTable VALUES
(?,?,?), (?,?,?), (?,?,?), (?,?,?), (?,?,?)';
ij> EXECUTE p1 USING 'VALUES (''1st'',1,1,''2nd'',2,2,''3rd'',
3,3,''4th'',4,4,''5th'',5,5)';
```

Mapping column values to return values

Multiple-row VALUES tables are useful in mapping column values to desired return values in queries.

```
-- get the names of all departments in OhioSELECT DeptName
FROM Depts,
(VALUES (1, 'Shoe'),
    (2, 'Laces'),
    (4, 'Polish'))
AS DeptMap(DeptCode,DeptDesc)
WHERE Depts.DeptCode = DeptMap.DeptCode
AND Depts.DeptLocn LIKE '%Ohio%'
```

You might also find it useful to store values used often for mapping in a persistent table and then using that table in the query.

Creating empty queries

You may need Derby to create "empty" queries in existing applications for filling in bits of functionality that Derby does not supply.

Empty queries of the right size and shape can be formed off a single values table and a "WHERE FALSE" condition:

```
SELECT *
FROM (VALUES ('',1,"TRUE")) AS ProcedureInfo(ProcedureName,NumParameters,

ProcedureValid)
WHERE 1=0
```

Localizing Derby

Derby offers support for locales.

The word *locale* in the Java platform refers to an instance of a class that identifies a particular combination of language and region. If a Java class varies its behavior according to *locale*, it is said to be locale-sensitive. Derby provides some support for locales for databases and other components such as the tools and the installer.

It also provides a feature to support databases in many different languages, a feature which is independent of a particular territory.

When you create or upgrade a database, you can use the territory attribute to associate a non-default territory with the database. For information about how to use the territory attribute, see the *Java DB Reference Manual*.

SQL parser support for Unicode

To support users in many different languages, Derby's SQL parser understands all Unicode characters and allows any Unicode character or number to be used in an identifier.

Derby does not attempt to ensure that the characters in identifiers are valid in the database's locale.

Character-based collation in Derby

A character set is a set of symbols and encodings. Character data types are represented as Unicode 2.0 sequences in Derby.

How collation works in Derby

Derby supports a wide range of character sets and encodes all of the character sets by using the Unicode support provided by the java.lang.Character class in the Java Virtual Machine (JVM) in which the Derby database runs. See the Java API documentation for the java.lang.Character class for the exact level of Unicode Standard that is supported.

A collation is a set of rules for comparing characters in a character set. In Derby the collation rules affect comparisons of the CHAR and VARCHAR data types. Collation rules also affect how the LIKE Boolean operator processes the CHAR, VARCHAR, CLOB, and LONG VARCHAR data types.

The default Derby collation rule is based on the binary Unicode values of the characters. So a character is greater than (<), equal to (=), or less than (>) another character based on the numeric comparison of the Unicode values. This rule allows for very efficient comparisons of character strings.

Note: When LIKE comparisons are used, Derby compares one character at a time for non-metacharacters. This is different than the way Derby processes = comparisons. The comparisons with the = operator compare the entire character string on left side of the = operator with the entire character string on the right side of the = operator. See the Differences between LIKE and equal (=) comparisons section below.

Territory-based collation

Derby also supports the ability to define collation rules that are appropriate to a territory, and is referred to as *territory-based collation*. Derby supports the territories that Java supports.

You can specifically set the territory of a database when you create the database. If you do not specify a territory, Derby uses the default territory of the JVM in which the database is created. Each JVM can support many territories that are independent from the default territory for the JVM. Collation support for these additional territories is provided through the `java.text.RuleBasedCollator` class and the set of rules for these territories. Refer to the JVM specification for details of how these rules are used to provide territory specific collation. Derby currently supports only running those rules that can be loaded dynamically from the running JVM based on the territory attribute. Overrides to these rules by the user are not supported.

The territory-based collation in Derby affects how the CHAR and VARCHAR data types are compared. Specifying territory-based collation also impacts how the LIKE Boolean operator processes CHAR, VARCHAR, CLOB, and LONG VARCHAR data.

Territory-based collation does add extra processing overhead to all character-based comparison operations.

Database attributes that control collation

When you create a Derby database, the attributes that you set determine the collation that is used with all of character data in the database. For example:

Table 13. The create database attributes that control collation

Example create commands	Collation is driven by
jdbc:derby:abcDB;create=true	Unicode codepoint collation (UCS_BASIC), which is the default collation for Derby databases.
jdbc:derby:abcDB;create=true;territory=es_	Unicode codepoint collation (UCS_BASIC). The `collation` attribute is not set.
jdbc:derby:abcDB;create=true;collation=TE	The territory of the JVM, since the `territory` attribute is not set. **Tip:** To determine the territory of the JVM, run `Locale.getDefault()`.
jdbc:derby:abcDB;create=true;territory=es_	The `territory` attribute.

Collation examples

With Unicode codepoint collation (UCS_BASIC), the numerical values of the Unicode encoding of the characters is used directly for ordering. For example, the FRUIT table contains the NAME column that uses the VARCHAR(20) data type. The contents of the NAME column are:

orange
apple
Banana
Pineapple
Grape

UCS_BASIC collation sorts all lower case letters before upper case letters. The statement `SELECT * FROM FRUIT ORDER BY NAME` returns:

apple
orange
Banana
Grape
Pineapple

If the database is created with the territory attribute set to en_US (English language, United States country code), and the collation attribute set to TERRITORY_BASED , the results of the statement `SELECT * FROM FRUIT ORDER BY NAME` returns:

apple
Banana
Grape
orange
Pineapple

The collation set for the database also impacts comparison operators on character data types. For example, the statement `SELECT * FROM FRUIT WHERE NAME > Banana` returns:

UCS_BASIC collation	Territory-based collation
Grape	Grape
Pineapple	orange
	Pineapple

Differences between LIKE and equal (=) comparisons

When you use territory-based collation, the comparisons can return different results when you use the LIKE and equal (=) operators. For example, suppose that the Derby database is set to use a territory where the character 'z' has same collation elements as 'xy'. Consider the following two WHERE clauses:

1. WHERE 'zcb' = 'xycb'
2. WHERE 'zcb' LIKE 'xy_b

For WHERE clause 1, Derby returns TRUE because the collation elements for the entire string 'zcb' will match the collation elements of the entire string 'xycb'.

For WHERE clause 2, Derby returns FALSE because collation element for character 'z' does not match the collation element for character 'x'. In addition, when metacharacter such as an underscore is used with the LIKE operator, the metacharacter counts for one character in the string value. A clause like WHERE 'xycb' LIKE '_cb' returns FALSE because 'x' is compared to the metacharacter _ and 'y' does not match 'c'.

Other components with locale support

Derby also provides locale support for the following components:

- Database error messages are in the language of the locale, if support is explicitly provided for that locale with a special library.

  For example, Derby explicitly supports Spanish-language error messages. If a database's locale is set to one of the Spanish-language locales, Derby returns error messages in the Spanish language.
- The Derby tools. In the case of the tools, locale support includes locale-specific interface and error messages and localized data display.

  For more information about localization of the Derby tools, see the *Java DB Tools and Utilities Guide*.

Localized messages require special libraries.

The locale of the database is set by the *territory=ll_CC* attribute when the database is created. However, the locale of the error messages and tools is not determined by the locale of the database. The locale of the error messages and tools is determined by the default system locale. This means that it is possible to create a database with a non-default locale. In such a case, error messages are not returned in the language of the locale of the database but are returned in the language of the default locale instead.

Note: You can override the default locale for ij with a property on the JVM. For more information, see the *Java DB Tools and Utilities Guide*.

Messages libraries

The following list describes the items required in order for Derby to provide localized messages.

- You must have the locale-specific Derby jar file. Derby provides such jars for only some locales. You will find the locale jar files in the */lib* directory in your Derby installation.
- The locale-specific Derby jar file must be in the classpath.

The locale-specific Derby jar file is named *derbyLocale_ll_CC.jar*, where *ll* is the two-letter code for language, and *CC* is the two-letter code for country. For example, the name of the jar file for error messages for the German locale is `derbyLocale_de_DE.jar.`

Derby supports the following locales:

- derbyLocale_cs.jar - Czech
- derbyLocale_de_DE.jar - German
- derbyLocale_es.jar - Spanish
- derbyLocale_fr.jar - French
- derbyLocale_hu.jar - Hungarian
- derbyLocale_it.jar - Italian
- derbyLocale_ja_JP.jar - Japanese
- derbyLocale_ko_KR.jar - Korean
- derbyLocale_pl.jar - Polish
- derbyLocale_pt_BR.jar - Brazilian Portuguese
- derbyLocale_ru.jar - Russian
- derbyLocale_zh_CN.jar - Simplified Chinese
- derbyLocale_zh_TW.jar - Traditional Chinese

Derby and standards

Derby adheres to SQL99 or newer standards wherever possible. This section describes those features currently in Derby that are not standard; these features are currently being evaluated and might be removed in future releases.

This section describes those parts of Derby that are non-standard or not typical for a database system.

ALTER TABLE syntax
: Derby uses a slightly different ALTER TABLE syntax for altering column defaults. While SQL99 uses DROP and SET, Derby uses DEFAULT.

Calling functions and procedures
: Derby supports the CALL (procedure) statement for calling external procedures declared by the CREATE PROCEDURE statement. Built-in functions and user-defined functions declared with the CREATE FUNCTION command can be called as part of an SQL select statement or by using either a VALUES clause or VALUES expression.

CLOB and BLOB data types
: Derby supports the standard CLOB and BLOB data types. BLOB and CLOB values are limited to a maximum of 2,147,483,647 characters.

Cursors
: Derby uses JDBC's result sets, and does not provide SQL for manipulating cursors except for positioned update and delete. Derby's scrollable insensitive cursors are provided through JDBC, not through SQL commands.

DECIMAL max precision
: For Derby, the maximum precision for DECIMAL columns is 31 digits. SQL99 does not require a specific maximum precision for decimals, but most products have a maximum precision of 15-32 digits.

Dynamic SQL
: Derby uses JDBC's Prepared Statement, and does not provide SQL commands for dynamic SQL.

Expressions on LONGs
: Derby permits expressions on LONG VARCHAR; however, LONG VARCHAR data types are not allowed in the following clauses, operations, constraints, functions, and predicates:

- GROUP BY clauses
- ORDER BY clauses
- JOIN operations
- PRIMARY KEY constraints
- Foreign KEY constraints
- UNIQUE key constraints
- MIN aggregate function
- MAX aggregate function
- [NOT] IN predicate
- UNION, INTERSECT, and EXCEPT operators

SQL99 also places some restrictions on expressions on LONG types.

Information schema
: Derby uses its own system catalog that can be accessed using standard JDBC DatabaseMetadata calls. Derby does not provide the standard Information Schema views.

NOT NULL characteristic
: The SQL standard says NOT NULL is a constraint, and can be named and viewed in the information schema as such. Derby does not provide naming for NOT NULL, nor

does it present it as a constraint in the information schema, only as a characteristic of the column.

Stored routines and PSM

Derby supports external procedures using the Java programming language. Procedures are managed using the CREATE PROCEDURE and DROP PROCEDURE statements.

Transactions

All operations in Derby are transactional. Derby supports transaction control using JDBC Connection methods. This includes support for savepoints and for the four JDBC transaction isolation levels. The only SQL command provided for transaction control is SET TRANSACTION ISOLATION.

XML data types and operators

Derby supports the XML data type and a set of operators that work with the XML data type, but does not provide JDBC support for the XML data type. The XML data type and operators are based on a small subset of the SQL/XML specification.

The XML data type and operators are defined only in the SQL layer.

There is no JDBC-side support for XML data types. It is not possible to bind directly into an XML value or to retrieve an XML value directly from a result set. Instead, you must bind and retrieve the XML data as Java strings or character streams by explicitly specifying the appropriate XML operator as part of the SQL statements:

- Use the XMLPARSE operator for binding data into XML values.
- Use the XMLSERIALIZE operator to retrieve XML values from a result set.

Additionally, there is no JDBC metadata support for the XML data type.

The XML data type is not allowed in any of the clauses or operations that are described in the section on expressions on LONG data types in Derby and standards.

For the XML operators to work properly, Derby requires that a JAXP parser, such as Apache Xerces, and Apache Xalan are included in the Java classpath. If either the parser or Xalan are missing from the classpath, Derby disallows any XML-related operations.

Classpath and version issues

Most Java virtual machines (JVMs) that are version 1.4 or later have a JAXP parser embedded in the JVM. If you are using one of these JVMs, you may not need to add any classes to your classpath. Some exceptions exist:

- In most version 1.4.2 JVMs, the version of Xalan that comes with the JVM is not new enough, so you must override the version of Xalan in the JVM with a newer version by using the Endorsed Standards Override Mechanism described at http://java.sun.com/j2se/1.4.2/docs/guide/standards/. To use this mechanism, download and install a binary distribution of Xalan from Apache and set the system property `java.endorsed.dirs` to point to the Xalan installation directory.
- In Sun JVMs later than version 1.4, Sun has renamed the JAXP packages so that Derby cannot access them. If you are using a Sun JVM later than version 1.4, download and install a binary distribution of Xalan from Apache and place the `xalan.jar` file in your classpath. The `xalan.jar` file automatically puts into the classpath the other required jars that are in the same directory.

Trademarks